Algorithmic Problems in Strings with Applications to the Analysis of Biological Sequences

Barton, Carl Samuel

Awarding institution:
King's College London

Algorithmic Problems in Strings with Applications to the Analysis of Biological Sequences

Carl Barton

A Thesis Submitted for the Degree of Doctor of Philosophy

Department of Informatics
King's College London

Acknowledgements

I would like to thank my supervisor, Prof. Costas Iliopoulos and second supervisor Prof. Maxime Crochemore for their support and friendship throughout my PhD. The many people in the Department of Informatics at Kings have offered support in various forms.

I am also grateful to all the people who have made contributions to my work but in particular my friend and collaborator Dr Solon P. Pissis who has been a constant throughout my PhD.

I would also like to thank my parents, Liz, Charlie, Caitlin, Max, Peter and all my other friends and family who supported me in my studies.

Abstract

Recent advances in molecular biology have dramatically changed the way biological data analysis is performed [119, 92]. *Next-Generation Sequencing* (NGS) technologies produce high-throughput data of highly controlled quality, hundreds of times faster and cheaper than a decade ago.

Mapping of short reads to a reference sequence is a fundamental problem in NGS technologies. After finding an occurrence of a high quality fragment of the read, the rest must be approximately aligned, but a good alignment would not be expected to contain a large number of *gaps* (consecutive insertions or deletions). We present an alternative alignment algorithm which computes the optimal alignment with a bounded number of gaps. Another problem arising from NGS technologies is merging overlapping reads into a single string. We present a data structure which allows for the efficient computation of the overlaps between two strings as well as being applicable to other problems.

Weighted strings are a representation of data that allows for a subtle representation of ambiguity in strings. In this document we present algorithms for other problems related to weighted strings: the computation of exact and approximate inverted repeats in weighted strings, computing repetitions and computing covers.

We investigate the average-case complexity of wildcard matching. Wildcards can be used to model single nucleotide polymorphisms and so, efficient algorithms to search for strings with wildcards are necessary. In this document we investigate how efficient algorithms for this problem can be on average.

There exist many organisms such as viruses, bacteria, eukaryotic cells, and archaea which have a circular DNA structure. If a biologist wishes to find occurrences of a particular virus in a carriers DNA sequence which may not be circular it must be possible to efficiently locate occurrences of circular strings. In this document we present a number of algorithms for circular string matching.

Publications

During the completion of this thesis the following related publications have been completed.

1. **Carl Barton**, Solon P. Pissis and Costas S. Iliopoulos. "Optimal Computation of all Tandem Repeats in a Weighted Sequence", *Algorithms for Molecular Biology*. 9(21), 2014.

2. **Carl Barton**, Costas S. Iliopoulos, Solon P. Pissis and William F. Smyth. "Fast & Simple Computations Using Prefix Tables under Hamming and Edit Distance", in *Combinatorial Algorithms*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014 (in press).

3. **Carl Barton**, Costas S. Iliopoulos and Solon P. Pissis, "Fast Algorithms for Approximate Circular String Matching", *Algorithms for Molecular Biology*, 9(9), 2014.

4. **Carl Barton** and Solon P. Pissis "Optimal Computation of all Repetitions in a Weighted String", in C. S. Iliopoulos, A. Langiu, editors, *2nd International Conference on Algorithms for Big Data (ICABD 2014)*, number 1146 in CEUR-WS Proceedings, pages 9-15, Aachen, 2014.

5. **Carl Barton**, Costas S. Iliopoulos and Solon P. Pissis, "Circular String Matching Revisited", in *Proceedings of the Fourteenth Italian Conference on Theoretical Computer Science (ICTCS 2013)*, pages. 200-205, 2013.

6. **Carl Barton**, Tomas Flouri, Costas S. Iliopoulos, and Solon P. Pissis, "GapsMis: Flexible Sequence Alignment with a Bounded Number of Gaps", in *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics (BCB 2013)*, pages 402:402-402:411, New York, NY, USA, 2013. ACM.

7. **Carl Barton**, Costas S. Iliopoulos, Nicola Mulder, and Bruce Watson. "Identification of all Exact and Approximate Inverted Repeats in Regular and Weighted Sequences", in *Proceedings of the International Conference on Engineering Applications of Neural Networks - 14th International Conference, EANN 2013, Halkidiki, Greece, September 13-16, 2013 Proceedings, Part II*, pages 11-19, 2013

8. **Carl Barton**, Costas S. Iliopoulos, Inbok Lee, Laurent Mouchard, Kunsoo Park and Solon Pissis. "Extending Alignments with k-mismatches and ℓ-gaps", *Theoretical Computer Science*, 525:80-88, 2014.

9. **Carl Barton**, Costas S. Iliopoulos and Solon P. Pissis. "Average-Case Optimal Approximate Circular String Matching", In A.-H. Dediu, E. Formenti, C. Martn-Vide, and B Truthe, editors, *Language and Automata Theory and Applications*, volume 8977 of Lecture Notes in Computer Science, pages 85-96. Springer Berlin Heidelberg, 2015

10. **Carl Barton** and Costas S. Iliopoulos. "On the Average-case Complexity of Pattern Matching with Wildcards", In preparation.

11. **Carl Barton**, Tomáš Flouri, Costas S. Iliopoulos and Solon P. Pissis. "Global and Local Sequence Alignment with a Bounded Number of Gaps", *Theoretical Computer Science*, 582(0):1-16, 2015, 2015.

Contents

1 Introduction

The study of biological sequences is a fundamental activity in many biological disciplines, it is an important task in areas such as epidemiology, comparative genomics, cancer genomics and many more. Within these fields the study of sequences is essential for a clear understanding of the mechanisms that cause disease, mutations and various biologically meaningful phenomena. DNA sequencing is the process of determining the order of nucleotides within DNA. DNA sequencing technologies include various methods that are used for determining the exact order of the nucleotide bases—adenine, guanine, cytosine, and thymine—in a DNA macromolecule.

Sequencing technologies have changed significantly in recent years. The traditional sequencing methods, developed in the mid 70's, had been the workhorse technology for DNA sequencing for almost thirty years. In 1977, the publication of two methodological papers by Sanger and Coulson on the rapid determination of DNA nucleotides [115, 117] would go on to revolutionise biology as a whole, providing a method for analysing complete genes and, later, entire genomes.

The method greatly improved earlier DNA sequencing techniques developed by Maxam and Gilbert [62] and Sanger and Coulson's own "plus and minus" method, published two years earlier [116]. Although a huge step forward in DNA sequencing methods, these technologies were very labour intensive. The sequencing of the human genome required many laboratories to cooperate for many years. Now, due to recent developments, the sequencing a human genome can be done in around two days.

The recent advances in sequencing technologies have dramatically changed the way biological data analysis is performed [119, 92]. *Next generation sequencing* (NGS) technologies produce high-throughput data of highly controlled quality, many times faster, and many many times cheaper, than a decade ago. These technologies show great promise from the biologi-

cal/medical viewpoints, opening up the possibility of personalised medicine and large scale genome studies. Traditional sequencing technologies produce large fragments of DNA in the order of 500 base pairs or more. In contrast NGS technologies produce DNA in short fragments as small as 25bps long in no particular order; meaning that we are left with a huge number of short fragments of the human genome, called *reads*. It is then necessary to reconstruct the full length genome from the reads. This is a particularly challenging problem as the same DNA fragments are sequenced many times to reduce the likelihood of errors and short read lengths increase the likelihood that multiple reads from different parts of the genome are identical.

There are two main methods of assembling a genome from a set of reads: *re-sequencing* with the help of a reference sequence and *de-novo* sequencing with no reference. The problem of *re-sequencing* consists of mapping the reads of the new genome against that of an existing genome of the same species. Many algorithms and programmes have been published to deal with the task of efficiently mapping millions of short sequences to a reference, namely Bowtie [82] BWA [85], SOAP2 [86], REAL [52]. *de-novo* sequencing offers a different set of challenges. Rather than attempting to map reads back to an existing genome, the main problem is to determine which reads are related by determining their overlaps to form *contigs* and from there combine contigs to form a *scaffolding*. The recent developments in NGS technologies have led to a proliferation of genomic data. The two major problems that have recently arisen due to this are the following:

1. Producing interesting data is now quicker than analysing it and interpreting the results.

2. The amount of produced data already exceeds the computers capacity, in terms of storing all the information in main memory or the time that is needed for properly processing it.

The scientific bottlenecks have moved from being able to produce interesting data for important societal health studies to being able to store, process and interpret the massive amount of data produced in numerous research centres. From the computer science viewpoint this raises a number of important algorithmic questions that fall into three broad areas:

- Genome Assembly: How can the massive amounts of data produced by DNA sequencers be efficiently and accurately interpreted both in the presence of a similar genome (*re-sequencing*) and without (*de-novo sequencing*).

- Storage and Querying: Storing and querying efficiently one sequence, or a limited set of sequences can be done using specialised data structures. But the ultimate goal is to be able to store not only one or several sequences, but hundreds or thousands, possibly more than 3 trillion base pairs.

- Sequence Analysis: How can we efficiently discover interesting patterns and structural features of a sequence, taking into account the types of errors present in NGS data.

The main areas considered in this thesis are algorithmic problems related to genome assembly and the analysis of sequences. The proliferation of biological data has increased the need for efficient algorithmic solutions to important biological problems. As the data increases so too does the benefit brought by efficient algorithmic solutions. In recent years the growth of publicly available sequences has been exponential, this, paired with the decrease in the costs of acquiring the sequences suggests that this will only continue.

Contribution [12, 15, 13]: In Chapter 3 we consider the task of extending short read alignments for the application of re-sequencing and show that it is possible to bound the number of gaps in both global and local alignments. Given the short length of the reads generated by most next generating sequencing platforms and the observed gap occurrence frequencies of organisms, the expected number of gaps in a read is very small. Due to the small number of gaps expected in a read it may be more realistic and preferred to take an alignment with no more than a small number of gaps, even if there exists a lower cost alignment with a higher number of gaps.

Traditional methods for sequence alignment cannot restrict the number of gaps, potentially reducing the quality of alignments. To increase the accuracy of short read alignment programs we present a new dynamic programming approach that runs in $\mathcal{O}(nk\ell)$ time where n is the length of the longest sequence, k is the allowed edit distance and ℓ is the number of gaps allowed. We then perform extensive experiments showing that bounding the number of gaps in an alignment increases the accuracy of the alignment when compared with traditional methods.

Contribution [21]: The computation of overlaps in strings is an important and fundamental step in processing data produced by many next generation sequencing technologies. Where a reference sequence is not present, the overlaps between reads are needed to rebuild the reference sequence. In this

situation all pairs of reads need to be checked against each other. However, where paired-end reads are used it may be possible that the paired reads overlap. Traditionally paired end reads are two reads sequenced from different ends of the same DNA fragment with an unknown middle section of approximately known length. New development have lead to the case where this unknown middle section does not exist and the two reads overlap. In this situation they must be merged if they are to be confidently used in a subsequent assembly. This means there is a need to consider how to efficiently compute the overlap between two strings.

In Chapter 4 we consider the computation of the prefix table, a fundamental data structure in string processing. We define and give algorithms for the computation of an approximate prefix table under both Hamming and edit distance. Additionally we outline applications to the longest approximate overlap problem for sequence assembly with paired end reads and, more generally, approximate string matching.

Contribution [16]: Inverted repeats are a feature of biological sequences which can have a number of important, biologically meaningful, functions. Inverted repeats can define the boundaries in transposons and indicate regions within a single sequence which can form base pairings with each other. These properties play an important role in genome instability and contribute not only to cellular evolution and genetic diversity but also to mutation and disease.

In Chapter 5 we consider the computation of exact and approximate inverted repeats under hamming distance in the context of weighted strings. The problem of detecting approximate inverted repeats has been well studied within the context of regular strings, but the weighted case is unexplored. In the process we develop a number of general techniques useful for algorithms on weighted strings. The algorithms run in $\mathcal{O}(n)$ time for exact repeats and $\mathcal{O}(kn)$ for the approximate inverted repeats.

Contribution [22, 19]: Repetitions or tandem repeats in biological sequences can be used as markers of genetic inheritance and to identify certain types of cancer. In Chapter 5 we consider the computation of tandem repeats in weighted strings. The computation of repetitions in weighted strings has been studied for a number of years and the best time complexity achieved so far is $\mathcal{O}(n^2)$ for all repetitions or $\mathcal{O}(d \log n)$ for the computation of repetitions of length d. In Chapter 5 we combine new and existing techniques to devise an $\mathcal{O}(n \log n)$ algorithm for the computation of repetitions in weighted

strings.

The first algorithm we present takes a significantly different approach when compared to existing algorithms. Previous algorithms directly modify the optimal algorithm for computing repetitions in normal strings or the partitioning algorithm of Karp, Miller and Rosenberg. Our algorithm does not take this approach and avoids an expensive pruning step which causes these algorithms to have a runtime of $\mathcal{O}(n^2)$. We then show an alternative optimal algorithm for computing repetitions in a weighted string and show how to modify this to compute covers, improving the next known algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. This algorithm takes a more traditional approach and we show that the partitioning technique used in [37] can be efficiently modified to work for weighted strings.

Contribution [14]: In Chapter 6 we consider the problem of pattern matching with wildcards and derive fast average-case algorithms. Pattern matching with wildcards can be used to find occurrences of sequences where single nucleotide polymorphisms or other ambiguity may be present. It is known that single nucleotide polymorphisms can cause the number of diseases and can affect the bodies response to pathogens, chemicals, drugs, vaccines, and other agents. Due to this efficient algorithms for the problem are of interest. It is also important to understand how fast algorithms of this nature can be. We investigate the average-case complexity of a number of wildcard matching problems, to the best of our knowledge this is the first study of this problem.

We consider variations of the problem of pattern matching with wildcards. For the problem of pattern matching with wildcards only in the text we derive an optimal $\mathcal{O}(\frac{n \log_\sigma m}{m})$ algorithm and where wildcards can appear only in the pattern or both we derive $\mathcal{O}(\frac{n(g+\log_\sigma m)}{m})$ algorithms for these problems where g is the number of wildcards in the pattern. We then show that these are optimal in a ubiquitous family of algorithms and then derive a general lower bound for the average-case complexity of pattern matching.

Contribution [17, 20, 18]: In Chapter 7 we consider the problem of approximate circular string matching. Many organisms from viruses, bacteria, archaea, mitochondrial DNA and plasmid DNA can have a circular DNA structure. Efficiently searching for these types of structures in larger biological sequences requires pattern matching algorithms specifically for this problem. Due to this it is important to design efficient algorithms to search for circular strings.

We present an expected linear time algorithm for exact matching; a fast

in practise and expected linear time algorithm for $k/m = \mathcal{O}(1/\log m)$ for k-mismatches, and an average optimal algorithm for k-differences. The average optimal algorithm has a search time of $\mathcal{O}(\frac{n(k+\log_\sigma m)}{m})$ for $k/m < 1/2 - \mathcal{O}(\frac{1}{\sqrt{\sigma}})$ and reduces the preprocessing time and space against the best known algorithm by a factor of at least $\mathcal{O}(\frac{m^2}{q})$ and $\mathcal{O}(m^2)$ respectively, where q is an appropriately chosen q-gram.

1.1 Outline of the Document

In Chapter 2 we give preliminaries and definitions necessary for the rest of the thesis; in Chapter 3 we introduce the problem of short read alignment and demonstrate why current algorithms are insufficient when dealing with gaps in reads; in Chapter 4 we consider problems related to merging paired end reads and provide efficient and practical algorithms for these and other tasks; in Chapter 5 we consider a number of problems related to algorithms on weighted string we consider the computation of following: exact and approximate inverted repeats, tandem repeats and covers; Chapter 6 considers the complexity of fast on average algorithms for pattern matching with wildcards; Chapter 7 we consider string matching in the situation where the pattern has a circular structure and present efficient solutions to approximate string matching problems and finally, in Chapter 8 we give concluding remarks.

2

Preliminaries etc

In this chapter we present the preliminaries and definitions required for the rest of the thesis. We focus only on the basic definitions of strings and weighted strings, extra definitions will be given where required in the appropriate chapter.

An *alphabet* Σ is a finite non-empty set of size σ, whose elements are called *letters*. In this thesis we use a number of different assumptions about the alphabet size. The most common assumption we make is that the alphabet size is constant, that is $\sigma = \mathcal{O}(1)$. Unless explicitly stated otherwise for the rest of the thesis the alphabet size should be assumed constant. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. We denote by $x[i]$, for all $0 \leq i < |x|$, the letter at index i of x. Each index i, for all $0 \leq i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the i-th letter of x is the letter at position $i - 1$ in x.

The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by xy. A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. Consider the strings x, y, u, and v, such that $y = uxv$, if $u = \varepsilon$, then x is a *prefix* of y, and if $v = \varepsilon$, then x is a *suffix* of y. Let x be a non-empty string and y be a string. We say that there exists an *occurrence* of x in y, or, more simply, that x *occurs in* y, when x is a factor of y. Let x and y be two strings on Σ, such that $|y| \geq |x|$ and $x = y[i \,..\, j]$, we say that x occurs at the *starting position* i in y.

A *wildcard* letter is a special letter, denoted by ϕ, that does not belong to alphabet Σ and matches with itself as well as with any letter of Σ. Two letters a and b such that $a, b \in \Sigma \cup \{\phi\}$ are said to *correspond* (denoted by

$a \approx^{\phi} b$) if they are equal or at least one of them is the wildcard letter.

A weighted string x on an alphabet Σ is a finite sequence of n sets. Every $x[i]$, for all $0 \leq i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having letter s_j at position i. Formally, $x[i] = \{(s_j, \pi_i(s_j)) | s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}$. A letter s_j occurs at position i of a weighted string x if and only if the *occurrence probability* of letter s_j at position i, $\pi_i(s_j)$, is greater than 0. A string u of length m is a factor of a weighted string if and only if it occurs at starting position i with *cumulative occurrence probability* $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. Given a *cumulative weight threshold* $1/z \in (0, 1]$, we say that factor u is *valid*, or equivalently that factor u has a valid occurrence, if it occurs at starting position i and $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$. For clarity of presentation, in the rest of this document, a set of ordered pairs in a weighted string is denoted by $[(s_0, \pi_i(s_0)), \ldots, (s_{\sigma-1}, \pi_i(s_{\sigma-1}))]$.

Example 1. *Let the following weighted string x and the cumulative weight threshold $1/z = 1/4$.*

Position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
x	`A`	`C`	`T`	`T`	*(`A`, 0.5)*	`T`	`C`	*(`A`, 0.6)*	`T`	`T`	`T`
					(`C`, 0.5)			*(`C`, 0.2)*			
					(`G`, 0.0)			*(`G`, 0.0)*			
					(`T`, 0.0)			*(`T`, 0.2)*			

`TGTCAT` is not a factor of x; `TATCCT` is a factor of x starting at position 3; and `TATCAT` is a valid factor of x starting at position 3 with cumulative occurrence probability 0.3.

For every string x and every natural number n, we define the n-th power of the string x, denoted by x^n, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$, for all $1 \leq k \leq n$. A string is said to be *primitive* if it cannot be written as v^e, where $e \geq 2$. A repetition in x is a non-trivial power of a primitive string occurring in x.

Formally, a *repetition* u^e, $e \geq 2$, in x is defined as a triple (i, p, e) such that: $u = x[i \,..\, i+p-1] = x[i+p \,..\, i+2p-1] = \ldots = x[i+(e-1)p \,..\, i+ep-1]$; u^{e+1} does not occur at position i; and u is primitive. A repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i-p]$. The integers p and e are called the *period* and the *exponent* of the repetition, respectively. In other words, a repetition is a primitively-rooted integer power u^e which is not followed by another occurrence of u; and a maximal repetition is a primitively-rooted integer power u^e which is not followed or preceded by another occurrence of u. If $e = 2$ the repetition is called *square*.

A *repetition* $v = u^e$, $e \geq 2$, in a weighted string x is defined as a quadruple (i, p, b, e) such that $u = v[0\,..\,p-1] = v[p\,..\,2p-1] = \ldots = v[(e-1)p\,..\,ep-1]$, where v is a factor of length ep of x occurring at position i, and each occurrence of u in v is a valid factor of x; u^{e+1} does not occur at position i; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. A repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i-p]$. The need for set b in uniquely defining a repetition can be seen in Example 2.

Example 2. *Let* $x = \mathtt{aab}[(\mathtt{a}, 0.5)(\mathtt{b}, 0.5)][(\mathtt{a}, 0.5)(\mathtt{b}, 0.5)]\mathtt{bab}$ *and* $1/z = 1/2$. *Then* $(1, 3, \{(2, \mathtt{a})\}, 2)$ *is a repetition in* x*, such that* $u = \mathtt{aba}$ *and* $v = \mathtt{abaaba}$.

The *Hamming distance* between two strings u and v, such that $|u| = |v| = n$, is the number of positions such that $u[i] \neq v[i]$ for $0 \leq i < n$. Given a non-negative integer k, we write $u \equiv_k^H v$ if the Hamming distance between u and v is at most k. Consider two weighted strings u and v such that $|u| = |v| = n$, we say they match with probability $1/z$ if u' is a valid factor of u and v' is a valid factor of v such that $|u'| = |u| = |v'| = |v| = n$ and $u' = v'$; similarly we say u and v match with at most k mismatches if $u' \equiv_k^H v'$.

Given a string x of length m and a string y of length $n \geq m$, the *edit distance* is the minimum total cost of operations required to transform one string into the other. For simplicity, we consider the cost of each to be 1 [84]. The allowed edit operations are as follows:

- *Insertion*: insert a letter in y, not present in the corresponding position in x; $(\varepsilon, b),\ b \neq \varepsilon$.
- *Deletion*: delete a letter in y, present in x; $(a, \varepsilon),\ a \neq \varepsilon$.
- *Substitution*: replace a letter in y with a letter in x; $(a, b),\ a \neq b, \mathtt{and}\ a, b \neq \varepsilon$.

We write $x \equiv_k^E y$ if the edit distance between x and y is at most k. Equivalently, if $x \equiv_k^E y$, we say that x and y have at most k *differences*. We refer to the *standard dynamic programming matrix* of x and y defined by $\mathsf{D}[i, 0] = i,\ \text{for } 0 \leq i \leq m,\ \mathsf{D}[0, j] = j,\ \text{for } 0 \leq j \leq n$, and for $1 \leq i \leq m, 1 \leq j \leq n$:

$$\mathsf{D}[i,j] = \min \begin{cases} \mathsf{D}[i-1, j-1] + 1 \ (\mathsf{if}\ x[i-1] \neq y[j-1]) \\ \mathsf{D}[i-1, j] + 1 \\ \mathsf{D}[i, j-1] + 1 \end{cases}$$

In this thesis sometimes we specifically consider the following alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. When considering this alphabet we now define which letters of the alphabet are equivalent under the complement relation ($\bar{=}$). The relation is defined by the following:

- $\mathsf{A} \bar{=} \mathsf{T}$.
- $\mathsf{C} \bar{=} \mathsf{G}$.
- $\mathsf{G} \bar{=} \mathsf{C}$.
- $\mathsf{T} \bar{=} \mathsf{A}$.

Given two strings u and v such that $|u| = |v| = n$ then $u \bar{=} v$ if and only if $u[i] \bar{=} v[i]$ for $0 \leq i < n$ and we denote the complement of a factor u by $\bar{u}$.

For a string x we denote by x^R the reverse of x. A factor w is an even *inverted repeat* if $w = u\bar{u}^R$ and an odd inverted repeat if $w = ua\bar{u}^R$ such that $u \in \Sigma^+$ and $a \in \Sigma$. In both cases we say the inverted repeat is of *radius* $|u|$. An inverted repeat $w = u\bar{u}^R$ is *centered* around j if and only if the start position of $\bar{u}^R$ is $j + 1$. A factor w is an even approximate inverted repeat under Hamming distance if $w = uv$ such that $v \equiv^k \bar{u}^R$ and an odd inverted repeat if $w = uav$ such that $v \equiv^k \bar{u}^R$ and $a \in \Sigma$. An inverted repeat can be represented as a triple (j, r, e) where j is the centre of the inverted repeat, r is the radius and $e = 1$ if the repeat is even.

Given a weighted string x an even (odd) weighted inverted repeat is a valid factor of x starting at i of the form $u\bar{u}^R$ (resp. $ua\bar{u}^R$) such that $\pi_i(u\bar{u}^R) \geq 1/z$ (resp. $\pi_i(ua\bar{u}^R) \geq 1/z$ and $a \in \Sigma$). An approximate even (odd) weighted inverted repeat is a valid factor of x starting at i of the form uv (resp. uav) such that $\pi_i(uv) \geq 1/z$ (resp. $\pi_i(uav) \geq 1/z$ and $a \in \Sigma$) and $v \equiv^k \bar{u}^R$. A weighted inverted repeat can be represented as a quadruple (j, r, e, b) where j is the centre of the inverted repeat, r is the radius $e = 1$ if the repeat is even and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$.

We denote by SA the *suffix array* of x, that is the array of length n of the starting positions of all sorted suffixes of x, i.e. for all $1 \leq r < n - 1$, we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$ [103]. Let $\mathsf{lcp}(r, s)$ denote the length of the longest common prefix of the words $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$, for all $0 \leq r, s < n-1$, and 0 otherwise. We denote by LCP the *longest common prefix* array of x defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$, for all $1 < r < n - 1$, and $\mathsf{LCP}[0] = 0$. The inverse iSA of the array SA is defined

by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n-1$. SA [103], iSA, and LCP [43] of x can be computed in time and space $\mathcal{O}(n)$.

A *trie* is a tree representing a collection of strings with one node per common prefix. That is it is the smallest tree such that:

- Each string is spelled out along some path starting at the root;
- Each edge is labelled with a character $c \in \Sigma$;
- A node has at most one outgoing edge labelled c, for $c \in \Sigma$.

The *suffix trie* of x is then defined as the trie of all the suffixes of x. We can now defined the *suffix tree* of x as the compressed suffix trie of x. By compressed we mean the resulting trie has only internal nodes of degree > 2 and the labels of the arcs consist of the concatenation of the chain of arcs that were removed.

In this chapter we have provided a number of basic definitions important for comprehending the rest of the thesis. We have focused on only the basic requirements and, where appropriate, additional definitions specific to a chapter will be given there.

3

Increased Accuracy Short Read Alignment

Alignments are a technique used to compare strings based on the notions of distance [84] or similarity between strings; for example, similarities among biological sequences [118]. Alignments are often computed incrementally through dynamic-programming-based algorithms [129].

A *gap* in an alignment is a maximal sequence of consecutive insertions or deletions (indels) of letters. Alignments have been heavily used in biological applications to compare biological sequences. It has been observed that rather than penalising all edit operations separately, it is desirable to penalise the formation of long gaps more severely than other operations [55].

A gap in a biological sequence can be described as the absence (resp. presence) of a region, which is (resp. is not) present in another sequence. Gaps are a naturally occurring feature of biological sequences. In many biological applications, a single mutational event can cause the insertion or deletion of an entire region (particularly in DNA), so the accurate detection of gaps in biological sequences is an important problem.

The creation of gaps in DNA sequences can be caused by a number of biological processes, long pieces of DNA can be copied and inserted by a single mutational event; slippage during the replication of DNA may cause the same area to be repeated multiple times as the replication machinery loses its place on the template; an insertion in one sequence paired with a reciprocal deletion in one other may be caused by unequal cross-over in meiosis; insertion of transposable elements—jumping genes—into a DNA sequence; insertion of DNA by retroviruses; and translocations of DNA between chromosomes [57].

In this chapter, we are directly motivated by the problem of *re-sequencing*—the assembly of a genome directed by a reference sequence. Due to recent developments in sequencing technologies (see [11, 70, 114], for example) se-

quencing an entire genome has become a routine procedure. Whole-genome sequencing creates masses of data (tens of gigabytes) in the form of short sequences (reads); and these reads must then be mapped (aligned) back to a reference sequence.

The performance of this procedure, in terms of speed, sensitivity, and accuracy, deteriorates in the presence of genomic variability and sequencing errors; particularly so for relatively short consecutive sequences of insertions or deletions. A plethora of short-read alignment programmes (e.g. Bowtie [82], SOAP2 [86], REAL [52], BWA [85], Bowtie2 [81]) have been published recently to address the task of mapping millions of short reads to a genome, focusing on various aspects of the procedure. In general, these tools allow for mismatches in the alignments, however, their ability to account for the insertion of gaps varies significantly and many perform poorly or not allowing for the insertion of gaps at all.

The *seed-and-extend* strategy [6] is applied in almost all short-read alignment programmes. After the fast alignment between a factor of the reference sequence and a *seed* (short high-quality prefix of the read, positions 0-3 in square brackets in Fig. 3.1) by a short-read alignment programme, an important problem is to find the alignment between a relatively short succeeding factor of the reference sequence and the remaining low-quality suffix of the read (positions 4-10 in Fig. 3.1). The extension of the alignment must allow for a number of mismatches (position 8 in Fig. 3.1) and the insertion of gaps (positions 4 and 7 in Fig. 3.1) in the factor of the reference sequence or in the read.

```
        [0 1 2 3] 4 5 6 7 8 9 10 11
Ref.    G C G A C G T C C G A A
        | | . |   | | |   . | |
Read    G C A A - G T - A G A
```

Figure 3.1: Alignment between the factor of the reference sequence, starting at position 0 and ending at position 10, and the read with two mismatches at positions 2 and 8 and two gaps of length one each inserted in the read at positions 4 and 7

The step that *extends* the seed alignment into a full alignment can either require that the read aligns *globally* (end-to-end), or *locally* (for instance, see [81]). From Fig. 3.1, in global alignment mode, it is clear to see that the insertion of a gap may be required in the leftmost position of the alignment (position 4 in Fig. 3.1), but we do not know the length of the succeeding

```
0 1 2 3 4 5 6 7 8 9 10
C G T C C G A A G T G
    | . | | | |
- - T A C G A A - - -
```

(a) Global alignment

```
2 3 4 5 6 7
T C C G A A
| . | | | |
T A C G A A
```

(b) Local alignment

```
0 1 2 3 4 5 6 7
C G T C C G A A
    | . | | | |
- - T A C G A A
```

(c) Semi-global alignment

Table 3.1: Alignments between $x =$ `CGTCCGAAGTG` and $y =$ `TACGAA`

factor of the reference sequence to be aligned beforehand. From this observation it becomes clear that an intermediate between the global and the local alignment is needed; this is known as *semi-global* alignment and it allows the insertion of a gap at the end of an alignment with no penalty, but penalises the insertion of a gap in the leftmost position of the alignment. In local alignment mode, an alignment that includes only a portion of the read (i.e. with some amount trimmed from one or both ends) with a high alignment score may be preferred over an end-to-end alignment with a lower alignment score. Traditionally there are two approaches for these problems: the Needleman-Wunsch algorithm [101] for semi-global alignment; and the Smith-Waterman algorithm [130] for local alignment. Both of these algorithms are based on the application of dynamic programming.

Example 3. *Let $x =$ `CGTCCGAAGT` and $y =$ `TACGAA`. Table 3.1 illustrates a global, a local, and a semi-global alignment between x and y, respectively.*

Although gaps may occur in large range of lengths, the short length of reads means that large gaps cannot be confidently detected. Fig. 3.2, shows the distribution of gap lengths in human exome sequencing[1]. This distribution agrees with other studies on gaps (cf. [102, 104, 120]). Fig. 3.2 represents a gap occurrence frequency of approximately 5.7×10^{-6} across the human *exome*—the part of the genome formed by exons, coding portions of genes in the genome that are expressed.

Fig. 3.2 shows that the occurrence of gaps decreases exponentially as the length increases. For short reads in the order of 25-150 base pairs (bp), the presence of a large number of gaps is very unlikely given the gap occurrence frequency, and introducing many gaps could reduce the mapping confidence of those reads. Therefore, applying a traditional dynamic programming ap-

[1]Data generated by the Exome Sequencing Programme at the NIHR Biomedical Research Centre at Guy's and St Thomas' NHS Foundation Trust in partnership with King's College London.

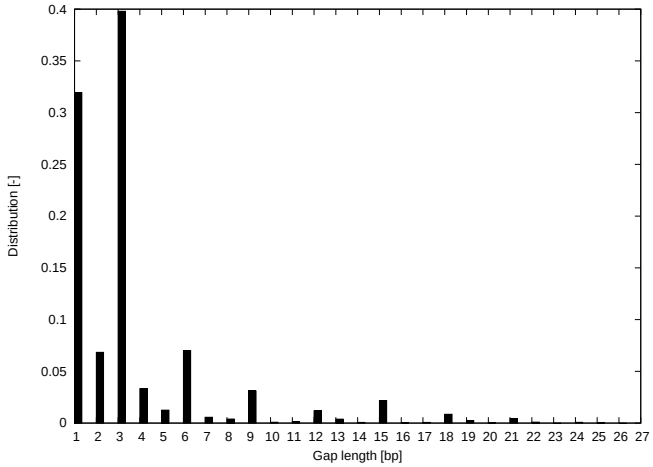


Figure 3.2: Distribution of gaps length in human exome sequencing

proach, which cannot *bound* the number of mismatches, insertions, and deletions in the alignment would greatly affect the mapping confidence.

Motivated by the aforementioned observations, in [46], algorithm `GapMis` was presented, an algorithm for pairwise global sequence alignment with a *single* gap. In this chapter, we present `GapsMis`, a generalisation of `GapMis`, that is, an algorithm for pairwise global sequence alignment with a variable, but bounded, number of gaps—solving the open problem stated in [47]. The algorithm requires time $\Theta(mk\ell)$ and space $\Theta(mk)$, where m is the length of the shortest sequence, k is the maximum allowed edit distance between the two sequences, and ℓ is the maximum allowed number of gaps inserted in the alignment. Here, we additionally present `GapsMis-L`, the analogous algorithm for pairwise local sequence alignment with a variable, but bounded, number of gaps. To test the accuracy of these algorithms, we have performed millions of pairwise sequence alignments, under realistic conditions based on the properties of real full-length genomes. The results show that `GapsMis` and `GapsMis-L` can increase the accuracy of extending short-read alignments compared to the traditional approaches.

The rest of this chapter is structured as follows. In Section 3.2, we present `GapsMis`, an algorithm based on dynamic programming to solve the former problem. By solving this problem, we mean the computation of a different version of the traditional dynamic programming matrix for global sequence alignment. The main difference is that it restricts the alignment such that it contains a variable, but bounded, number of gaps. In Section 3.3, we present `GapsMis-L`, the analogous algorithm for pairwise local sequence alignment. In Section 3.4, we describe our implementation of algorithms `GapsMis` and

`GapsMis-L` as a programme for pairwise semi-global and local sequence alignment with scoring matrices and affine gap penalty scores; and we present extensive experimental results. Finally, we briefly conclude with some final remarks and future proposals in Section 3.5.

Our Contribution. In this chapter we present an alternative sequence alignment algorithm for extending short read alignments. Our algorithm allows for the bounding of the number of gaps inserted in the alignment. We then show that this functionality increases the accuracy of the resulting alignments.

3.1 Problem Definitions

An *aligned pair* is a pair (a, b) such that $(a, b) \in \Sigma \cup \{\varepsilon\} \times \Sigma \cup \{\varepsilon\}/\{\varepsilon, \varepsilon\}$. An *alignment* between string x and string y is a string of aligned pairs whose projection on the first component is x and the projection on the second component is y. Let $\delta_E(x, y)$, defined for two strings x and y, denote the minimum number of edit operations required to transform one string into the other. Moreover, let $\delta_E^l(x, y)$ denote the minimum number of edit operations required to transform x into y, such that their alignment that corresponds to these edit operations consists of at most ℓ gap sequences. First we define some notions of alignment

A *gap sequence*, or simply *gap*, is a finite non-empty maximal sequence of length p of aligned pairs

$$(a_0, b_0), (a_1, b_1), \ldots, (a_{p-1}, b_{p-1}),$$

such that *either*

- $a_0 = a_1 =, \ldots, = a_{p-1} = \varepsilon$ *or*
- $b_0 = b_1 =, \ldots, = b_{p-1} = \varepsilon$ holds.

A *gap-free sequence* is a string of length p of aligned pairs

$$(a_0, b_0), (a_1, b_1), \ldots, (a_{p-1}, b_{p-1}),$$

such that $a_i, b_i \in \Sigma$, for all $0 \leq i < p$.

The aforementioned ideas are the basis of the pairwise *global sequence alignment with k-differences and l-gaps* problem, formally defined as follows.

Problem 4 (Global Sequence Alignment with k-differences and l-gaps)**.** *Given a string x of length n, a string y of length $m \leq n$, an integer k, such that $0 \leq k < n$, and an integer ℓ, such that $0 \leq \ell \leq k$, find a prefix of x, say x', such that $\delta_E^\ell(x', y)$ is minimum, $\delta_E^\ell(x', y) \leq k$, and for the corresponding alignment $z = z_0 g_0 z_1 g_1 \ldots g_{\beta-1} z_\beta$: $\beta \leq \ell$; z_0 is a possibly empty gap-free sequence; $g_0, \ldots, g_{\beta-1}$ are gap sequences; and $z_1, \ldots, z_\beta$ are non-empty gap-free sequences.*

Finding a prefix of x that satisfies the above restrictions is equivalent to the notion of semi-global alignment between x and y, which is relevant to the application of re-sequencing.

Example 5. *Let $x =$ GCGACGTCCGAA, $y =$ GCAAGTAGA, $k = 4$, and $\ell = 2$. Consider the following alignment between the prefix $x' = x[0\,..\,10] =$ GCGACGTCCGA of x and y.*

```
0 1 2 3 4 5 6 7 8 9 10
G C G A C G T C C G A
| |   | |   | |   |
G C A A - G T - A G A
```

With the above problem definition, this alignment is a solution to this problem instance, since $\delta_E^\ell(x', y)$ is minimum, $\delta_E^\ell(x', y) = 4 \leq k$, $\beta = 2 \leq \ell$, positions 0-3 are z_0, position 4 is g_0, positions 5-6 are z_1, position 7 is g_1, and positions 8-10 are z_2.

Let $\mathsf{G}_s[0\,..\,n, 0\,..\,m]$ be a matrix, for all $1 \leq s \leq \ell$, where $\mathsf{G}_s[i, j]$ contains the minimum number of operations required to transform factor $x[0\,..\,i-1]$ of x into factor $y[0\,..\,j-1]$ of y allowing for the insertion of at most s gaps. More formally, $\mathsf{G}_s[i, j] = \delta_E^s(x[0\,..\,i-1], y[0\,..\,j-1])$.

In order to compute the exact location of the inserted gaps, we also need to maintain matrix $\mathsf{H}_s[0\,..\,n, 0\,..\,m]$, such that

$$\mathsf{H}_s[i, j] = \begin{cases} -b & \text{if a gap of length } b \text{ is inserted after } y[j-1] \\ a & \text{if a gap of length } a \text{ is inserted after } x[i-1] \\ 0 & \text{if no gap is inserted} \end{cases}$$

The computation of matrix H_s, for all $1 \leq s \leq \ell$, denotes the direction of the gap inserted. The direction of the gap is identified by defining insertions in y as negative integers and insertions in x as positive.

Example 6. *Let $x =$ ACATCGACG and $y =$ CATTCGACG. Table 3.2 illustrates matrix G_1 and matrix H_1, respectively. Table 3.3 illustrates matrix G_2 and matrix H_2, respectively.*

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	0	1	2	3	4	5	6	7	8	9
1	A	1	1	1	3	4	5	6	6	8	9
2	C	2	1	2	2	4	4	6	7	6	9
3	A	3	3	1	3	3	5	5	6	8	7
4	T	4	4	4	1	3	4	5	6	7	8
5	C	5	4	5	5	2	3	4	5	6	7
6	G	6	6	5	6	5	3	3	4	5	6
7	A	7	7	6	6	6	5	4	3	4	5
8	C	8	7	8	7	7	6	5	4	3	4
9	G	9	9	8	9	8	7	6	5	4	3

(a) Matrix G_1

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
1	A	1	0	0	0	0	0	0	0	0	0
2	C	2	0	0	0	0	0	0	0	0	0
3	A	3	0	0	0	0	0	0	0	0	0
4	T	4	0	0	0	0	0	-2	0	0	-5
5	C	5	0	0	0	0	0	-1	-2	0	-4
6	G	6	0	0	0	2	0	0	-1	-2	0
7	A	7	0	0	0	3	2	0	0	-1	-2
8	C	8	0	0	0	0	0	2	1	0	-1
9	G	9	0	0	0	0	4	0	2	1	0

(b) Matrix H_1

Table 3.2: Matrix G_1 and matrix H_1 for $x =$ ACATCGACG and $y =$ CATTCGACG

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	0	1	2	3	4	5	6	7	8	9
1	A	1	1	1	3	4	5	6	6	8	9
2	C	2	1	2	2	4	4	6	7	6	9
3	A	3	3	1	2	3	4	5	6	7	7
4	T	4	4	4	1	2	3	4	5	6	7
5	C	5	4	5	2	2	2	4	5	5	7
6	G	6	6	5	3	3	3	2	4	5	5
7	A	7	7	6	4	4	4	4	2	4	5
8	C	8	7	8	5	5	4	5	4	2	4
9	G	9	9	8	6	6	6	4	5	4	2

(a) Matrix G_2

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	**0**	-1	-2	-3	-4	-5	-6	-7	-8	-9
1	A	**1**	0	0	0	0	0	0	0	0	0
2	C	2	**0**	0	0	0	0	0	0	0	0
3	A	3	0	**0**	**-1**	0	-3	0	0	-6	0
4	T	4	0	0	0	**0**	-2	-3	-4	-5	-6
5	C	5	0	0	1	0	**0**	0	0	0	0
6	G	6	0	0	2	0	0	**0**	-1	-2	0
7	A	7	0	0	3	0	0	0	**0**	-1	-2
8	C	8	0	0	4	0	0	0	1	**0**	-1
9	G	9	0	0	5	0	0	0	2	1	**0**

(b) Matrix H_2

Table 3.3: Matrix G_2 and matrix H_2 for $x =$ ACATCGACG and $y =$ CATTCGACG

Instead of considering a global alignment between x and y, it may be more appropriate to determine the best alignment between a factor of x and a factor of y. The notion of distance is not appropriate for stating this question. Indeed, when we try to minimise a distance, the factors that lead to the smallest values are the factors that occur simultaneously in the two strings x and y, factors that may be reduced to just a few letters. We thus rather utilise a notion of similarity between strings, for which equalities between letters are positively valued, and mismatches, insertions, and deletions are negatively valued. The search for a similar factor consists then in maximising a quantity representative of the similarity between the strings.

To measure the degree of similarity between two strings x and y, we utilise a scoring function. This function, denoted by Sub_s, measures the degree of resemblance between two letters of the alphabet. The larger the value $\mathsf{Sub}_s(a, b)$ is, the more similar the two letters a and b are. We assume that the function satisfies $\mathsf{Sub}_s(a, a) > 0$, for $a \in \Sigma$, and $\mathsf{Sub}_s(a, b) < 0$, $a, b \in \Sigma$ with $a \neq b$.

The function Sub_s is symmetrical, but it is not a distance since it does not satisfy the conditions of positivity, separation, nor the triangle inequality. Indeed, we can attribute different scores to several equalities of letters: we can have $\mathsf{Sub}_s(a, a) \neq \mathsf{Sub}_s(b, b)$. This allows a better control of the equalities that are more greatly desired. The insertion and deletion functions must also be negatively valued (their values are integers): $\mathsf{Ins}_s(b) < 0$ and $\mathsf{Del}_s(a) < 0$, for $a, b \in \Sigma$.

We define then the similarity $\mathsf{sim}(x, y)$ between the strings x and y by $\mathsf{sim}(x, y) = \max\{\text{score of } \sigma : \sigma \in \Sigma_{x,y}\}$, where $\Sigma_{x,y}$ is the set of sequences of edit operations transforming x into y. The score of an element $\sigma \in \Sigma_{x,y}$ is the sum of the scores of the edit operations of σ. An *optimal* local alignment between two strings x and y is a pair of strings (u, v), for which u is a factor of x and v is a factor of y, and $\mathsf{sim}(u, v)$ is maximum.

Let $\mathsf{sim}^\ell(x, y)$, defined for two strings x and y, denote the maximum score of operations required to transform one string into the other, such that their alignment consists of at most ℓ gap sequences.

The aforementioned ideas are the basis of the *local sequence alignment with l-gaps* problem, formally defined as follows.

Problem 7 (Local Sequence Alignment with l-gaps)**.** *Given a string x of length n, a string y of length $m \leq n$, and an integer ℓ, such that $0 \leq \ell < n$, find a pair of strings (u, v), for which u is a factor of x and v is a factor of y, such that $\mathsf{sim}^\ell(u, v)$ is maximum, and for the corresponding alignment*

$z = z_0 g_0 z_1 g_1 \dots g_{\beta-1} z_\beta$: $\beta \leq \ell$; $g_0, \dots, g_{\beta-1}$ are gap sequences; and $z_0, \dots, z_\beta$ are non-empty gap-free sequences.

Let $\mathsf{S}_s[0\,..\,n, 0\,..\,m]$ be a matrix, for all $0 \leq s \leq \ell$, where $\mathsf{S}_s[i,j]$ contains the similarity score between some suffix u of $x[0\,..\,i-1]$ and some suffix v of $y[0\,..\,j-1]$ such that this score is maximum and the alignment between u and v contains at most s gaps. If this score is negative, then we set $\mathsf{S}_s[i,j] = 0$. More formally, $\mathsf{S}_s[i,j] = \max(\{\mathsf{sim}^s(x[q\,..\,i-1], y[p\,..\,j-1]) : 0 \leq q \leq i \text{ and } 0 \leq p \leq j\} \cup \{0\})$.

In order to compute the exact location of the inserted gaps, we also need to maintain matrix $\mathsf{H}_s[0\,..\,n, 0\,..\,m]$, similarly as for the case of Problem 4.

3.2 Algorithm GapsMis

Algorithm `GapsMis` is a generalisation of algorithm `GapMis`, first introduced in [47], for solving Problem 4. Algorithm `GapsMis` computes matrices $\mathsf{G}_{1..\ell}$ and matrices $\mathsf{H}_{1..\ell}$. It takes as input the string x of length n and the string y of length m. It assumes that matrices G_1 and H_1 are computed by algorithm `GapMis`.

Proposition 8 ([47])**.** *Algorithm `GapMis` correctly computes matrix G_1 and matrix H_1 in time $\Theta(mn)$.*

Theorem 9. *Algorithm `GapsMis` correctly computes matrices $\mathsf{G}_{2..\ell}$ and matrices $\mathsf{H}_{2..\ell}$ in time $\Theta(mn\ell)$.*

Proof. Let $\mathsf{G}_s[0,j] = j$ and $\mathsf{G}_s[i,0] = i$ for all $1 < s \leq \ell$, $0 \leq i \leq n$ and $0 \leq j \leq n$. Without loss of generality, assume that we want to compute cell $\mathsf{G}_s[i,j]$, for all $1 < s \leq \ell$, $0 \leq i \leq n$, $0 \leq j \leq m$.

Let $u = \mathsf{G}_{s-1}[r,j] + i - r$, such that

$$r \in \{0, \dots, i-1\} : \mathsf{G}_{s-1}[r,j] + i - r \leq \mathsf{G}_{s-1}[c,j]$$

for all, $0 \leq c \leq i$.

Let $v = \mathsf{G}_{s-1}[i,q] + j - q$, such that

$$q \in \{0, \dots, j-1\} : \mathsf{G}_{s-1}[i,q] + j - q \leq \mathsf{G}_{s-1}[i,c]$$

for all, $0 \leq c \leq j$.

Further, let $w = \mathsf{G}_{s-1}[i-1, j-1] + \delta_E(x[i-1], y[j-1])$.

ALGORITHM `GapMis`(x, n, y, m)

{Where x is a string of length n and y is a string of length $m < n$.}
Initialising matrix G_1 and matrix H_1
for $i \leftarrow 0$ to n **do**
  $\mathsf{G}_1[i, 0] \leftarrow i$;
  $\mathsf{H}_1[i, 0] \leftarrow i$;
end for
for $j \leftarrow 0$ to m **do**
  $\mathsf{G}_1[0, j] \leftarrow j$;
  $\mathsf{H}_1[0, j] \leftarrow -j$;
end for
{Computing matrix G_1 and matrix H_1}
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to m **do**
    if $i < j$ **then**
      $u \leftarrow \mathsf{G}_1[i-1, j-1] + \delta_E(x[i-1], y[j-1])$;
      $v \leftarrow \mathsf{G}_1[i, i] + (j - i)$;
      $\mathsf{G}_1[i, j] \leftarrow \mathsf{min}\{u, v\}$;
      if $v < u$ **then**
        $\mathsf{H}_1[i, j] \leftarrow i - j$;
      else
        $\mathsf{H}_1[i, j] \leftarrow 0$;
      end if
    end if
    if $i > j$ **then**
      $u \leftarrow \mathsf{G}_1[i-1, j-1] + \delta_E(x[i-1], y[j-1])$;
      $v \leftarrow \mathsf{G}_1[j, j] + (i - j)$;
      $\mathsf{G}_1[i, j] \leftarrow \mathsf{min}\{u, v\}$;
      if $v < u$ **then**
        $\mathsf{H}_1[i, j] \leftarrow i - j$;
      else
        $\mathsf{H}_1[i, j] \leftarrow 0$;
      end if
    end if
    if $i = j$ **then**
      $\mathsf{G}_1[i, j] \leftarrow \mathsf{G}_1[i-1, j-1] + \delta_E(x[i-1], y[j-1])$;
      $\mathsf{H}_1[i, j] \leftarrow 0$;
    end if
  end for
end for
return G_1 and H_1;

The idea of our algorithm is to iteratively compute the minimum cost alignment with at most ℓ gaps by allowing the insertion of one gap each time we compute the matrix G_s. The first gap insertion is handled by algorithm `GapMis`. In order to allow for the insertion of the s-th gap we consider the matrix G_{s-1}, computed for the $s-1$-th gap, and take the minimum cost alignment from $\mathsf{G}_{s-1}[0,j]$ to $\mathsf{G}_{s-1}[i,j]$ (lines 15-16 in Algorithm `GapsMis`), from $\mathsf{G}_{s-1}[i,0]$ to $\mathsf{G}_{s-1}[i,j]$ (lines 20-21 in Algorithm `GapsMis`), along with the possibility of extending the alignment from $\mathsf{G}_s[i-1,j-1]$ to $\mathsf{G}_s[i,j]$ (line 24 in Algorithm `GapsMis`). Each of these three costs is added to the respective cost of forming the new alignment (for example, the cost of any inserted gap), separately, forming variables u, v, and w. We then take the minimum value of these three, which is the new minimum cost alignment with at most s gaps (line 25 in Algorithm `GapsMis`). Taking these minima in a naive way would lead to a poor runtime of $\Theta(mn^2)$ per matrix, however, we can improve this to $\Theta(mn)$ by reusing the previous minima we have already computed.

If a new gap is inserted then the indices r and q must be computed before we can find out the new minimum cost alignment. We will now outline how to reduce the complexity of computing r to constant time; q can be computed similarly.

Assume we compute $\mathsf{G}_s[i,j]$ after we have computed $\mathsf{G}_s[i-1,j]$. Clearly to compute $\mathsf{G}_s[i-1,j]$ we must have computed the following.

$$r \in \{0,\ldots,i-1\} : \mathsf{G}_{s-1}[r,j] + i - r \leq \mathsf{G}_{s-1}[c,j]$$

for all, $0 \leq c \leq i-1$.

If we store the value of r we compute for $\mathsf{G}_s[i-1,j]$, then we can easily update the new value of r for $\mathsf{G}_s[i,j]$ as follows.

$$\textbf{if } \mathsf{G}_{s-1}[i,j] < \mathsf{G}_{s-1}[r,j] + i - r \textbf{ then } r \leftarrow i$$

The above check can easily be done in constant time. In algorithm `GapsMis` we store array minI, which maintains the current minimum value for each column. Clearly the same optimisation can be made for q (minJ in algorithm `GapsMis`); however we only need to store one value for q at a time as we compute the matrix row by row.

The effect of the above optimisation means all computation at each cell will take constant time. This reduces the time complexity of the algorithm to $\Theta(mn)$ per matrix.

Trivially, the computation of $\mathsf{H}_s[i,j]$ depends only on the minimum value selected from $\{u, v, w, \mathsf{G}_{s-1}[i,j]\}$ (lines 31-43 in Algorithm `GapsMis`).

$$\mathsf{H}_s[i,j] = \begin{cases} \mathsf{H}_{s-1}[i,j] & \text{if } \mathsf{G}_{s-1}[i,j] = \mathsf{G}_s[i,j] \\ i - r & \text{if } u = \mathsf{G}_s[i,j] \\ -(j-q) & \text{if } v = \mathsf{G}_s[i,j] \\ 0 & \text{if } w = \mathsf{G}_s[i,j] \end{cases}$$

Hence, algorithm `GapsMis` correctly computes matrices $\mathsf{G}_{2..\ell}$ and $\mathsf{H}_{2..\ell}$ in time $\Theta(mn)$ per matrix, so $\Theta(mn\ell)$ in total. □

For solving Problem 4, it is sufficient to locate a value in $\mathsf{G}_{1..\ell}[0\,..\,n, m]$, say $\mathsf{G}_s[i,m]$, for some $0 \leq i \leq n$, such that $\mathsf{G}_s[i,m]$ is minimum and $\mathsf{G}_s[i,m] \leq k$. Starting the trace-back from cell $\mathsf{H}_s[i,m]$ determines the corresponding alignment if there is no j, $j < s \leq \ell$, such that $\mathsf{G}_s[i,m] = \mathsf{G}_j[i,m]$.

Algorithm `GapsPos` determines this alignment by finding the positions of the inserted gaps. It takes as input the matrix H_s, an integer $0 \leq i \leq n$, the length m of y, and the maximum allowed number s of inserted gaps. It produces as output the exact number $\beta \leq s$ of inserted gaps and the following three arrays:

- Array gap_pos of size s, such that gap_pos[i], for all $0 \leq i \leq s-1$, gives the position of the i-th inserted gap or 0 if no gap is inserted.
- Array gap_len of size s, such that gap_len[i], for all $0 \leq i \leq s-1$, gives the length of the i-th inserted gap.
- Array where of size s, such that if gap_len[i] > 0, then where$[i]$, for all $0 \leq i \leq s-1$, is equal to 1 if the i-th gap is inserted in y or equal to 2 if the i-th gap is inserted in x.

Example 10. *Let x = ACATCGACG, y = CATTCGACG, $k = 2$, and $\ell = 2$. Starting the trace-back from cell $\mathsf{H}_2[9,9]$ (see Table 3.4a in bold), i.e. $i = 8$, gives a solution since $\mathsf{G}_2[9,9] = 2$ is minimum and $\mathsf{G}_2[9,9] = 2 \leq k$ (see Table 3.4b). Finally, we can determine the corresponding alignment by finding the positions of the inserted gaps, which are at $\mathsf{H}_2[3,3]$ and $\mathsf{H}_2[0,1]$ ($\beta = 2 \leq \ell$), using algorithm `GapsPos`. A solution to this problem instance is the alignment in Fig. 3.3.*

However, since the threshold k is given, pruned versions $\mathsf{G}^{\mathrm{P}}_{1..\ell}$ and $\mathsf{H}^{\mathrm{P}}_{1..\ell}$ of matrices $\mathsf{G}_{1..\ell}$ and matrices $\mathsf{H}_{1..\ell}$, respectively, can be computed in time $\Theta(mk)$ per matrix, similarly as shown in [57] for computing the traditional dynamic programming matrix for global sequence alignment.

```
ALGORITHM GapsMis(x, n, y, m)
    {Where x is a string of length n and y is a string of length m < n.}
    G_1, H_1 ← GapMis(x, n, y, m);
    for s ← 2 to ℓ do
      for i ← 0 to n do
 5:     G_s[i, 0] ← i;
        H_s[i, 0] ← i;
      end for
      for j ← 0 to m do
        G_s[0, j] ← j;
10:     H_s[0, j] ← −j;
      end for
    end for
    for s ← 2 to ℓ do
      minI[0 . . m] ← 0;
15:   for i ← 1 to n do
        minJ ← 0;
        for j ← 1 to m do
          H_s[i, j] ← 0;
          newminI ← 0;
20:       if G_{s−1}[i, j] < G_{s−1}[minI[j], j] + i − minI[j] then
            minI[j] ← i;
            newminI ← 1;
          end if
          u ← G_{s−1}[minI[j], j] + i − minI[j];
25:       newminJ ← 0;
          if G_{s−1}[i, j] < G_{s−1}[i, minJ] + j − minJ then
            minJ ← j;
            newminJ ← 1;
          end if
30:       v ← G_{s−1}[i, minJ] + j − minJ;
          w ← G_s[i − 1, j − 1] + δ_E(x[i − 1], y[j − 1]);
          G_s[i, j] ← min{u, v, w};
          if u = min{u, v, w} and newminI = 1 then
            H_s[i, j] ← i − minI[j];
35:       else
            H_s[i, j] ← H_{s−1}[i, j];
          end if
          if v = min{u, v, w} and newminJ = 1 then
            H_s[i, j] ← −(j − minJ);
40:       else
            H_s[i, j] ← H_{s−1}[i, j];
          end if
        end for
      end for
45: end for
    return G_{1..ℓ} and H_{1..ℓ};
```

ALGORITHM `GapsPos`(H_s, i, m, s)
{Where H_s is an array, i, m and s integer.
Initialising variables.}
$j \leftarrow m$;
$\beta \leftarrow 0$;
$\mathsf{gap\_pos}[0\,..\,s-1] \leftarrow 0$;
$\mathsf{gap\_len}[0\,..\,s-1] \leftarrow 0$;
$\mathsf{where}[0\,..\,s-1] \leftarrow 0$;
{Trace-back}
while $i \geq 0$ **and** $j \geq 0$ **do**
  if $\mathsf{H}_s[i,j] = 0$ **then**
    $i \leftarrow i-1$;
    $j \leftarrow j-1$;
  else
    if $\mathsf{H}_s[i,j] < 0$ **then**
      $\mathsf{gap\_pos}[\beta] \leftarrow j$;
      $\mathsf{gap\_len}[\beta] \leftarrow -\mathsf{H}_s[i,j]$;
      $\mathsf{where}[\beta] \leftarrow 1$;
      $j \leftarrow j + \mathsf{H}_s[i,j]$;
    else
      $\mathsf{gap\_pos}[\beta] \leftarrow i$;
      $\mathsf{gap\_len}[\beta] \leftarrow \mathsf{H}_s[i,j]$;
      $\mathsf{where}[\beta] \leftarrow 2$;
      $i \leftarrow i - \mathsf{H}_s[i,j]$;
    end if
    $\beta \leftarrow \beta + 1$;
  end if
end while
return $\beta, \mathsf{gap\_pos}, \mathsf{gap\_len}, \mathsf{where}$;

Lemma 11. *There exist at most $k+1$ cells of matrix G_s, $1 \leq s \leq \ell$, that give a solution to Problem 4.*

Proof. Consider the cell $\mathsf{G}_s[i,m]$, for some $0 \leq i \leq n$. Since the edit distance can be at most k, only those paths which are never more than k away from the main diagonal can provide a valid solution, and as we wish to align prefixes of x with y, only cells $\mathsf{G}_s[m-k\,..\,m,j]$ can possibly provide a valid solution, and thus $m-k \leq i \leq m$. There exist at most $k+1$ such cells. □

Hence we only need to compute a diagonal band of width $2k+1$ in

```
0 1 2 3 4 5 6 7 8 9
A C A - T C G A C G
  | |   | | | | | |
- C A T T C G A C G
```

Figure 3.3: Solution for $x = \texttt{ACATCGACG}$, $y = \texttt{CATTCGACG}$, $k = 2$, and $\ell = 2$

matrices $\mathsf{G}_{1..\ell}$ and in matrices $\mathsf{H}_{1..\ell}$. As a result, algorithm `GapsMis` can be easily modified to compute $\mathsf{G}^{\mathrm{P}}_{1..\ell}$ and $\mathsf{H}^{\mathrm{P}}_{1..\ell}$ in time $\Theta(mk\ell)$, by replacing lines 11 and 13 of algorithm `GapsMis` by the following.

```
for i ← 1 to min{n, m + k} do
  for j ← max{1, i − k} to min{m, i + k} do
    ...
  end for
end for
```

The same can be applied to algorithm `GapMis` [47]. For any cell providing a valid solution, algorithm `GapsPos` requires an additional time of $\mathcal{O}(m)$.

The computation of matrices G_s and H_s, for all $2 \leq s \leq \ell$, depends only on matrix G_{s-1}. Trivially, the space complexity can be reduced to $\Theta(mk)$. Therefore, we obtain the following result.

Theorem 12. *Problem 4 can be solved in time $\Theta(mk\ell)$ and space $\Theta(mk)$.*

Example 13. *Let $x = \texttt{ACATCGACG}$, $y = \texttt{CATTCGACG}$, $k = 2$, and $\ell = 2$. Table 3.4 illustrates matrix $\mathsf{G}^{\mathrm{P}}_2$ and matrix $\mathsf{H}^{\mathrm{P}}_2$, respectively.*

Alternatively, we could compute matrices $\mathsf{G}^{\mathrm{P}}_{1..\ell}$ and matrices $\mathsf{H}^{\mathrm{P}}_{1..\ell}$ based on a simple *alignment scoring* scheme depending on the application of the algorithm (see Section 7.5, for example), and compute the *maximum score* among all possible alignments of x and y in time $\Theta(k)$ by Lemma 11.

3.3 Algorithm GapsMis-L

Algorithm `GapsMis-L` is a modification of algorithm `GapsMis` for solving Problem 7. Algorithm `GapsMis-L` computes matrices $\mathsf{S}_{0..\ell}$ and matrices $\mathsf{H}_{0..\ell}$. It takes as input the string x of length n and the string y of length m.

The primary difference between Algorithm `GapsMis-L` and Algorithm `GapsMis` is the computation of two additional matrices, S_0 and H_0, and a

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	0	1	2							
1	A	1	1	1	3						
2	C	2	1	2	2	4					
3	A		3	1	2	3	4				
4	T			4	1	2	3	4			
5	C				2	2	2	4	5		
6	G					3	3	2	4	5	
7	A						4	4	2	4	5
8	C							5	4	2	4
9	G								5	4	2

(a) Matrix $\mathsf{G}_2^{\mathrm{P}}$

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	A	T	T	C	G	A	C	G
0	ϵ	0	-1	-2							
1	A	1	0	0	0						
2	C	2	0	0	0	0					
3	A		0	0	-1	0	-3				
4	T			0	0	0	-2	-3			
5	C				1	0	0	0	0		
6	G					0	0	0	-1	-2	
7	A						0	0	0	-1	-2
8	C							0	1	0	-1
9	G								2	1	0

(b) Matrix $\mathsf{H}_2^{\mathrm{P}}$

Table 3.4: Matrix $\mathsf{G}_2^{\mathrm{P}}$ and matrix $\mathsf{H}_2^{\mathrm{P}}$ for $x =$ ACATCGACG, $y =$ CATTCGACG, and $k = 2$

different method of computing matrices S_1 and H_1. We first present a new algorithm to compute S_0 and H_0 which computes the minimum cost alignment with no gaps. From here, we do not compute matrices S_1 and H_1 with Algorithm GapMis, but in the same way as matrices $2, \ldots, s$ in Algorithm GapsMis. Then the computation of all other matrices is the same as Algorithm GapsMis.

Intuitively, the reason for the computation of the additional matrices S_0 and H_0 is due to the nature of local alignment. When considering the optimal semi-global alignment with no gaps, the only possible solution is given by the cells $\mathsf{G}_0[i, j]$ such that $i = j$. The same is not true for local alignments, as we do not require that the entire string is aligned, there are many solutions and we do not know their lengths *a priori*. Due to this, we are required to determine the zero gap alignments explicitly.

From here, we assume that matrices S_0 and H_0 are computed by algorithm NoGap-L. A negative value for the insertion (or deletion) of $i > 0$ letters is denoted by function $\mathsf{Gap}(i)$.

Proposition 14. *Algorithm NoGap-L correctly computes matrix S_0 and matrix H_0 in time $\Theta(mn)$.*

Proof. Trivial. □

Theorem 15. *Algorithm GapsMis-L correctly computes matrices $\mathsf{S}_{1..\ell}$ and matrices $\mathsf{H}_{1..\ell}$ in time $\Theta(mn\ell)$.*

ALGORITHM `NoGap-L`(x, n, y, m)
  $\{x$ is a string of length n and y is a string of length $m < n.\}$
  {Initialising matrix S_0 and matrix H_0}
  for $i \leftarrow 0$ to n **do**
    $\mathsf{S}_0[i, 0] \leftarrow 0$;
    $\mathsf{H}_0[i, 0] \leftarrow 0$;
  end for
  for $j \leftarrow 0$ to m **do**
    $\mathsf{S}_0[0, j] \leftarrow 0$;
    $\mathsf{H}_0[0, j] \leftarrow 0$;
  end for
  {Computing matrix S_0 and matrix H_0}
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to m **do**
      $\mathsf{S}_0[i, j] \leftarrow \mathsf{max}\{\mathsf{S}_0[i-1, j-1] + \mathsf{Sub}_s(x[i-1], y[j-1]), 0\}$;
      $\mathsf{H}_0[i, j] \leftarrow 0$;
    end for
  end for
  return S_0 and H_0;

Proof. Similar to the proof of Theorem 9. □

For solving Problem 7, it is sufficient to locate a largest value in matrix S_s. We trace back then the path from the position of this value by going up in matrix H_s similarly as for the case of Problem 4. We stop the scan, in general, on a null value. Trivially, the space complexity can be reduced to $\Theta(mn)$. Therefore, we obtain the following.

Theorem 16. *Problem 7 can be solved in time* $\Theta(mn\ell)$ *and space* $\Theta(mn)$.

3.4 Experimental Results

We implemented algorithms `GapsMis` and `GapsMis-L` as a programme to compute the optimal semi-global and local alignment between two sequences with a variable, but bounded, number of gaps. We applied a simple alignment scoring scheme for DNA (resp. protein) sequences, that uses the scoring matrix NUC.4.4 [97] (resp. BLOSUM62 [98]) to assign scores for every possible nucleotide (resp. residue) match or mismatch; and affine gap penalty to score the insertion of a gap. The penalty for a gap of length $i > 0$ is computed as

```
ALGORITHM GapsMis-L(x, n, y, m)
    {x is a string of length n and y is a string of length m < n.}
    S_0, H_0 ← NoGap-L(x, n, y, m);
    for s ← 1 to ℓ do
      for i ← 0 to n do
 5:     S_s[i, 0] ← 0;
        H_s[i, 0] ← 0;
      end for
      for j ← 0 to m do
        S_s[0, j] ← 0;
10:     H_s[0, j] ← 0;
      end for
    end for
    for s ← 1 to ℓ do
      maxI[0 . . m] ← 0;
15:   for i ← 1 to n do
        maxJ ← 0;
        for j ← 1 to m do
          H_s[i, j] ← 0;
          newmaxI ← 0;
20:       if S_{s−1}[i, j] + Gap(0) > S_{s−1}[maxI[j], j] + Gap(i − maxI[j]) then
            maxI[j] ← i;
            newmaxI ← 1;
          end if
          u ← S_{s−1}[maxI[j], j] + Gap(i − maxI[j]);
25:       newmaxJ ← 0;
          if S_{s−1}[i, j] + Gap(0) > S_{s−1}[i, maxJ] + Gap(j − maxJ) then
            maxJ ← j;
            newmaxJ ← 1;
          end if
30:       v ← S_{s−1}[i, maxJ] + Gap(j − maxJ);
          w ← S_s[i − 1, j − 1] + Sub_s(x[i − 1], y[j − 1]);
          S_s[i, j] ← max{u, v, w, 0};
          if u = max{u, v, w, 0} then
            H_s[i, j] ← i − maxI[j];
35:       else
            H_s[i, j] ← H_{s−1}[i, j];
          end if
          if v = max{u, v, w, 0} then
            H_s[i, j] ← −(j − maxJ);
40:       else
            H_s[i, j] ← H_{s−1}[i, j];
          end if
        end for
      end for
45: end for
    return S_{0..ℓ} and H_{0..ℓ};
```

		0	1	2	3	4	5	6	7	8	9	10	11	12	13
		ϵ	G	G	G	A	A	T	C	T	A	C	C	C	T
0	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	5	5	0	0	0	5	0	0	0	0
2	G	0	5	5	5	0	1	1	0	0	0	1	0	0	0
3	G	0	5	10	10	1	0	0	0	0	0	0	0	0	0
4	G	0	5	10	15	6	5	5	5	5	5	5	5	5	5
5	T	0	0	1	6	11	2	10	1	10	1	1	1	1	10
6	C	0	0	0	5	2	7	0	15	5	6	6	6	6	5
7	T	0	0	0	5	1	0	12	5	20	10	10	10	10	11
8	A	0	0	0	5	10	6	2	8	10	25	15	15	15	15
9	G	0	5	5	5	1	6	2	5	10	15	21	11	11	11
10	G	0	5	10	10	1	0	2	5	10	15	11	17	7	7
11	G	0	5	10	15	6	5	5	5	10	15	11	7	13	5
12	C	0	0	1	6	11	2	2	10	10	15	20	16	12	9
13	C	0	0	0	5	2	7	2	7	10	15	20	25	21	10
14	C	0	0	0	5	1	0	3	7	10	15	20	25	30	17

(a) Matrix S_2

		0	1	2	3	4	5	6	7	8	9	10	11	12	13
		ϵ	G	G	G	A	A	T	C	T	A	C	C	C	T
0	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	G	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0
3	G	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0
4	G	0	0	0	**0**	0	**-2**	-3	-4	-5	-6	-7	-8	-9	-10
5	T	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0
6	C	0	0	0	2	0	0	0	**0**	-1	0	0	0	0	-6
7	T	0	0	0	3	0	0	0	1	**0**	-1	-2	-3	-4	0
8	A	0	0	0	4	0	0	1	0	1	**0**	-1	-2	-3	-4
9	G	0	0	0	0	0	0	0	3	2	1	0	0	0	0
10	G	0	0	0	0	0	0	0	4	3	2	0	0	0	0
11	G	0	0	0	0	0	-2	-3	-4	4	**3**	0	0	0	-10
12	C	0	0	0	0	0	0	5	0	5	4	**0**	0	0	0
13	C	0	0	0	9	0	0	6	0	6	5	0	**0**	0	-2
14	C	0	0	0	10	0	0	0	0	7	6	0	0	**0**	0

(b) Matrix H_2

Table 3.5: Matrix S_2 and matrix H_2 for $x =$ AGGGTCTAGGGCCC and $y =$ GGGAATCTACCT.

$$\textit{gap opening penalty} + (i-1) \times \textit{gap extension penalty}.$$

The total score of each alignment is obtained by adding these two scores, and the optimal alignment is the one with the maximum total score. The same alignment scoring scheme is applied in package EMBOSS [111].

Example 17. *Let* $x =$ AGGGTCTAGGGCCC, $y =$ GGGAATCTACCT, $\mathsf{Sub}_s(a,a) = 5$, *for* $a \in \Sigma$, $\mathsf{Sub}_s(a,b) = -4$, $a, b \in \Sigma$ *with* $a \neq b$, *gap opening penalty* $= 10$, *and gap extension penalty* $= 0$. *Table 3.5 illustrates matrix* S_2 *and matrix* H_2, *respectively. The largest value in matrix* S_2 *is* $\mathsf{S}_2[14,12] = 30$. *We trace back then the path from the position of this value,* $\mathsf{H}_2[14,12]$, *by going up in matrix* H_2 *(see matrix* H_2 *in bold). We stop the scan on* $\mathsf{S}_2[2,1]$. *We obtain the following alignment.*

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
G  G  G  -  -  T  C  T  A  G  G  G  C  C  C
|  |  |           |  |  |  |              |  |  |
G  G  G  A  A  T  C  T  A  -  -  -  C  C  C
```

GapsMis and GapsMis-L were implemented in the C programming language, and were developed under the GNU/Linux operating system. The

programme takes as input arguments two files with the sequences (DNA or protein) in FASTA format (or in plain text), and then produces an EMBOSS-like text file with the optimal semi-global or local alignment as output. The user can also provide the maximum allowed number ℓ of inserted gaps with the modifier `-l <int>` and the maximum allowed edit distance k between the two sequences with the modifier `-k <int>`.

Species	Length of queries [bp]	Gap occurrence frequency	GapMis	gm -f 2	gm -f 3	gm -onf 2	gm -onf 3	needle	GapsMis -l 2	GapsMis -l 3
AT	100	2.4×10^{-5}	999,099	998,404	997,561	999,207	999,259	999,126	999,519	**999,524**
AT	150	2.4×10^{-5}	998,805	998,171	997,542	999,024	999,152	999,115	999,498	**999,510**
BV	100	1.7×10^{-5}	999,361	998,868	998,229	999,432	999,459	999,353	**999,628**	**999,628**
BV	150	1.7×10^{-5}	999,196	998,771	998,249	999,347	999,432	999,378	**999,649**	**999,649**
HS	100	5.7×10^{-6}	999,809	999,615	999,419	999,822	999,825	999,782	**999,871**	**999,871**
HS	150	5.7×10^{-6}	999,795	999,606	999,408	999,817	999,825	999,793	**999,882**	**999,882**

Table 3.6: Correct alignments using `GapMis`, `gapmis_one_to_one_f`, `gapmis_one_to_one_onf`, `needle`, and `GapsMis`. Each of the datasets consists of 1,000,000 pairs of sequences; the highest number of correct alignments for each dataset is indicated in bold.

In order to evaluate the performance of `GapsMis`, we compared its performance to the respective performance of the following:

- `GapMis` [47], a programme for pairwise semi-global sequence alignment with a *single* gap.
- `gapmis_one_to_one_f`, a function of the library `libgapmis` [5]. Function `gapmis_one_to_one_f` provides the user the option to split the query sequence into f fragments, based on the observed gap occurrence frequency and the query length, by taking the number of fragments as input argument. It then uses algorithm `GapMis` as a black box to identify a single gap in each fragment independently. The total score of the alignment is obtained by adding the f individual scores of the fragments. We denote this function by `gm -f <int>`, where `<int>` is the number of fragments f used as input argument.
- `gapmis_one_to_one_onf` [5], a function of the library `libgapmis`. Function `gapmis_one_to_one_onf` computes the alignment by using the optimal number of fragments. First, it takes the maximum allowed number of fragments as input argument, say $f_{\max}$, and only computes the total score of the alignments, for each different number $1, 2, \ldots, f_{\max}$ of fragments. It then uses function `gapmis_one_to_one_f` to compute the alignment by passing the optimal number of fragments—the one that

gives the maximum total score in the previous step—as input argument. We denote this function by `gm -onf <int>`, where `<int>` is the maximum number of fragments $f_{\max}$ used as input argument.

- EMBOSS `needle`, which implements Needleman-Wunsch algorithm for semi-global alignment. The Needleman-Wunsch algorithm is the traditional approach used for semi-global alignment. `needle` is currently one of the most popular pairwise sequence alignment programmes for semi-global alignment.

In order to evaluate the performance of `GapMis`, `gapmis_one_to_one_f`, `gapmis_one_to_one_onf`, `needle`, and `GapsMis` under real conditions, we simulated $1,000,000$ 100 bp-long query sequences from the 30 Mbp chromosome 1 of *Arabidopsis thaliana* (AT) obtained from [99] and inserted mismatches and gaps into the reference sequence; we then aligned these sequences back against the original reference sequence using these programmes. As mismatch occurrence frequency and gap occurrence frequency we have used the ones observed in AT [93], these are 1.6×10^{-3} and 2.4×10^{-5}, respectively. The distribution of insertions and deletions are 42% of the inserted gaps were insertions and 58% deletions, this is due to insertions being less common than deletions and also the observed frequency in AT [93]. With respect to the lengths of the gaps that have been inserted, the distribution of gap lengths shown in Fig. 3.2 was used; this is consistent with other studies on gap distributions (cf. [102, 104, 120]).

An effort was made in all cases to run the programmes in a similar way and a similar environment to ensure a valid comparison. The gap opening penalty was set to 10 and the gap extension penalty to 0.5—the default values in the EMBOSS package. The experimental setup is as follows: we consider an alignment as *valid* if the number of inserted gaps is less or equal to those that have been inserted. Furthermore, an alignment is considered *correct* when the total length of gaps inserted is smaller or equal to the length of the gaps that have been inserted *and* the number of mismatches is less than or equal to the mismatches inserted. Since the queries were simulated, it was possible for us to know where in the reference sequence each query was extracted. Hence, we were able to classify each generated alignment as valid/invalid and correct/incorrect. Finally, we define *accuracy* as the proportion of correct alignments in the dataset.

Thus, we evaluated the accuracy of the aforementioned programmes in *extending* an alignment end-to-end, assuming that the *seed* part of the alignment is already performed by using a conventional indexing scheme, that is,

Programme	Species	Length of queries [bp]	Gap occurrence frequency	Gap opening penalty	Gap extension penalty	Valid alignments	Correct alignments
`needle`	AT	100	2.4×10^{-5}	10	0.5	99,988	99,917
`needle`	AT	100	2.4×10^{-5}	15	0.5	99,992	99,911
`needle`	AT	100	2.4×10^{-5}	20	0.5	99,996	99,850
`GapsMis -l 2`	AT	100	2.4×10^{-5}	10	0.5	99,997	**99,956**
`GapsMis -l 3`	AT	100	2.4×10^{-5}	10	0.5	99,997	**99,956**
`needle`	AT	150	2.4×10^{-5}	10	0.5	99,991	99,919
`needle`	AT	150	2.4×10^{-5}	15	0.5	99,992	99,901
`needle`	AT	150	2.4×10^{-5}	20	0.5	99,996	99,834
`GapsMis -l 2`	AT	150	2.4×10^{-5}	10	0.5	100,000	**99,957**
`GapsMis -l 3`	AT	150	2.4×10^{-5}	10	0.5	100,000	**99,957**

Table 3.7: Valid and correct alignments using `needle` and `GapsMis`. Each of the datasets consists of 100,000 pairs of sequences; the highest number of correct alignments for each dataset is indicated in bold.

a hash-based index [52] or an FM index [85][2]. We repeated the same experiments with 150 bp-long query sequences and using other gap occurrence frequencies—observed in *Beta vulgaris* (BV) [93] and *Homo sapiens* (HS) exome [120].

The high accuracy of `GapsMis` is demonstrated by the results shown in Table 3.6. The results show that `GapsMis` has the highest accuracy in all cases. `GapsMis` can increase the accuracy of extending short-read alignments end-to-end by 0.01-0.04% compared to `needle`. Given the observed gap occurrence frequencies, the increased accuracy of gap identification is significant. For instance, the proportion of pairs of sequences with gaps in the six datasets of Table 3.6 ranged from 0.85% to 3.5%. The accurate identification of gaps is shown to be fundamental in various studies on disorders; for example, on Hajdu-Cheney syndrome, a disorder of severe and progressive bone loss [120].

`needle` cannot, by design, guarantee the insertion of a bounded number of gaps into the alignment with the default values for the gap opening penalty. Theoretically one could further increase the gap opening penalty in `needle` until this is guaranteed. However, this would have a potentially catastrophic impact on accuracy as mismatches would become overly preferred.

This intuition was checked by conducting the following experiment. We obtained $100,000$ 100 bp-long and $100,000$ 150 bp-long query sequences from the 30 Mbp chromosome 1 of AT and inserted mismatches and gaps into the

[2]Notice that comparing the proposed method to state-of-the-art short-read aligners is not relevant here, as we do not wish to evaluate the entire short-read alignment pipeline but rather focus *only* on the extension part of the alignments.

Programme	Species	Length of queries [bp]	Gap occurrence frequency	Gap opening penalty	Gap extension penalty	Valid alignments	Correct alignments
`needle`	BV	200	1.7×10^{-5}	10	0.5	99,997	99,931
`needle`	BV	200	1.7×10^{-5}	15	0.5	99,998	99,927
`needle`	BV	200	1.7×10^{-5}	20	0.5	99,999	99,889
`GapsMis -l 2`	BV	200	1.7×10^{-5}	10	0.5	99,999	99,956
`GapsMis -l 3`	BV	200	1.7×10^{-5}	10	0.5	99,999	**99,957**
`needle`	HS	250	5.7×10^{-6}	10	0.5	99,998	99,977
`needle`	HS	250	5.7×10^{-6}	15	0.5	99,999	99,975
`needle`	HS	250	5.7×10^{-6}	20	0.5	100,000	99,959
`GapsMis -l 2`	HS	250	5.7×10^{-6}	10	0.5	99,999	**99,983**
`GapsMis -l 3`	HS	250	5.7×10^{-6}	10	0.5	99,999	**99,983**

Table 3.8: Valid and correct alignments using `needle` and `GapsMis`. Each of the datasets consists of 100,000 pairs of sequences; the highest number of correct alignments for each dataset is indicated in bold.

reference sequence; then we aligned them back against the original reference sequence using `needle` and `GapsMis` similar to the previous experiment. In `needle`, the gap opening penalty ranged from 10 to 20, and the gap extension penalty was set to 0.5. In `GapsMis`, the gap opening penalty was always set to 10 and the gap extension penalty to 0.5.

Our assumption is confirmed by the results shown in Table 3.7. Increasing the gap opening penalty increases the valid alignments, but has a negative impact on the accuracy of `needle`. On the other hand, `GapsMis` guarantees higher accuracy with the default values. The proportion of pairs of sequences with gaps in the two datasets of Table 3.7 were 2.41% and 3.63%. We repeated the same experiment by simulating 200 and 250 bp-long query sequences using other gap occurrence frequencies. The higher accuracy of `GapsMis` becomes evident by the results shown in Table 3.8. The proportion of pairs of sequences with gaps in the two datasets of Table 3.8 were 3.44% and 1.46%.

In order to evaluate the efficiency of `GapsMis`, we compared its performance to the respective performance of `needle`. We thus avoided the unfair comparison of these two programmes against the well-optimised library functions of `libgapmis`. We simulated $10,000$ pairs of 100, 150, 200, and 250 bp-long query sequences from AT, similarly to the above experiment. Two versions of `GapsMis` were used: one with the modifiers `-l 3 -k 30` to set $\ell = 3$, $k = 30$, and use the $\Theta(mk\ell)$-time algorithm; and one with only the modifier `-l 3` to use the $\Theta(mn\ell)$-time algorithm. As it is demonstrated by the results in Fig. 3.4, `GapsMis` was able to complete the assignment much faster than `needle`. The version with the modifier `-k 30` was always the fastest, confirming our theoretical results.

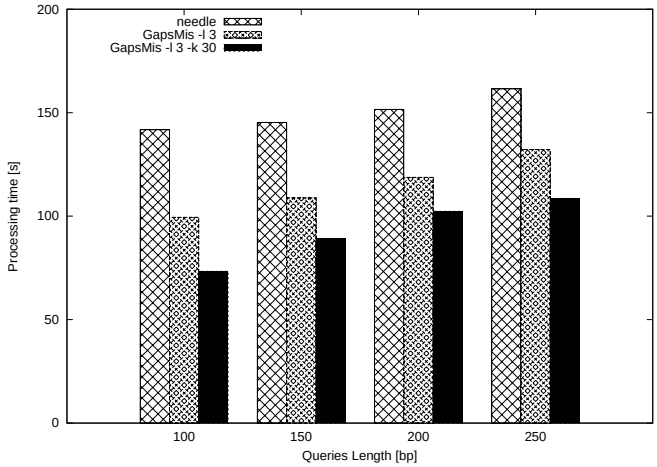


Figure 3.4: Processing time of `needle` and `GapsMis` for aligning $10,000$ pairs of sequences.

In order to evaluate the performance of `GapsMis-L`, we compared its performance to the respective performance of EMBOSS `water`, which implements Smith-Waterman algorithm for local alignment. The Smith-Waterman algorithm is the traditional approach used for local alignment. `water` is, up-to-date, one of the most popular pairwise sequence alignment programmes for local alignment.

In order to evaluate the performance of `water` and `GapsMis-L` under real conditions, we simulated $1,000,000$ 150 bp-long query sequences from the 30 Mbp chromosome 1 of AT and inserted mismatches and gaps into the reference sequence; then we aligned them back against the original reference sequence using these programmes. As mismatch occurrence frequency and gap occurrence frequency we used 1.6×10^{-3} and 2.4×10^{-5}, respectively—the ones observed in AT. Since, in practice, insertions occur less frequently than deletions, 42% of the inserted gaps were insertions and 58% deletions—also observed in AT. For the length of the inserted gaps, we used the distribution of gap lengths shown in Fig. 3.2. In each case, an effort was made to run the programmes in an as similar way as possible to ensure a fair comparison. The gap opening penalty was set to 10 and the gap extension penalty to 0.5—the default values in the EMBOSS package. Thus, we evaluated the accuracy of the aforementioned programmes in *extending* an alignment locally, assuming that the *seed* part of the alignment is already performed by using a conventional indexing scheme. We repeated the same experiment by simulating 200 bp-long query sequences and using other gap occurrence frequencies—observed in BV and HS exome.

The high accuracy of `GapsMis-L` is demonstrated by the results shown in Table 3.9. The results show that `GapsMis-L` has the highest accuracy in all cases.

Species	Length of queries [bp]	Gap occurrence frequency	`water`	`GapsMis-L -l 3`
AT	150	2.4×10^{-5}	999,668	**999,694**
AT	200	2.4×10^{-5}	999,675	**999,697**
BV	150	1.7×10^{-5}	999,747	**999,781**
BV	200	1.7×10^{-5}	999,753	**999,782**
HS	150	5.7×10^{-6}	999,909	**999,931**
HS	200	5.7×10^{-6}	999,903	**999,920**

Table 3.9: Correct alignments using `water` and `GapsMis-L`. Each of the datasets consists of 1,000,000 pairs of sequences; the highest number of correct alignments for each dataset is indicated in bold.

In order to evaluate the efficiency of `GapsMis-L`, we compared its performance to the respective performance of `water`. We simulated $10,000$ pairs of 100, 150, 200, and 250 bp-long query sequences from AT, similarly to the above experiments. As it is demonstrated by the results in Fig. 3.5, `GapsMis-L` was able to complete the assignment much faster than `water`.

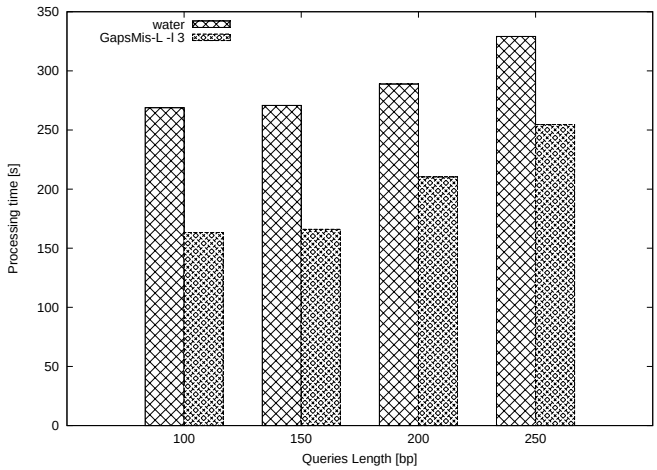


Figure 3.5: Processing time of `water` and `GapsMis-L` for aligning $10,000$ pairs of sequences.

The experiments were conducted on a Desktop PC using 1 Intel i7 2600 CPU at 3.4 GHz core under Linux. `GapsMis` is distributed under the GNU

General Public License (GPL). Our implementation is available at `http://www.inf.kcl.ac.uk/research/projects/gapmis/`, which is set up for maintaining the source code and the man page documentation. We also make available, on the same website, all the scripts used to generate the aforementioned datasets for reproducibility.

3.5 Conclusions and Future Work

In this chapter, we have presented `GapsMis`, an algorithm for pairwise global sequence alignment with a variable, but bounded, number of gaps. Algorithm `GapsMis` is based on the computation of a variant of the traditional dynamic programming matrix for global sequence alignment. The algorithm requires time $\Theta(mk\ell)$ and space $\Theta(mk)$, where m is the length of the shortest sequence, k is the maximum allowed edit distance between the two sequences, and ℓ is the maximum allowed number of gaps inserted in the alignment. Additionally, we presented Algorithm `GapsMis-L`, the analogous algorithm for pairwise local sequence alignment with a variable, but bounded, number of gaps.

These new algorithms are motivated by the next-generation re-sequencing application. We have demonstrated that `GapsMis` and `GapsMis-L` are more suitable and efficient than traditional approaches for extending short-read alignments. The increased flexibility provided by bounding the number of gaps inserted in an alignment increases the accuracy when compared to the traditional sequence alignment scheme of scoring matrices and affine gap penalty scores. The presented experimental results are very promising, both in terms of gap identification and efficiency; this suggests that further research and development of `GapsMis` and `GapsMis-L` is desirable.

For future work, we plan on doing the following: first, due to the massive amount of data produced by the next-generation sequencing technologies, develop accelerated SSE- and GPU-based library versions of `GapsMis` and `GapsMis-L` (see [5, 4], for example); and integrate the functions of this library into a short-read alignment pipeline.

4

Computing Approximate Prefix Tables and Their Applications

The computation of overlaps between reads is a fundamental problem in a number of sequencing tasks. One method to quickly sequence DNA is by a technique known as *shotgun sequencing*. In shotgun sequencing DNA is randomly broken up into lots of small fragments; these small fragments are then sequenced to obtain reads. To reduce the chance of the sequencing machine introducing errors this process is repeated multiple times, resulting in a set of reads with a high *coverage*. From this process each nucleotide is represented in many overlapping reads. The overlapping ends of different reads can be used to assemble a continuous sequence.

Due to the high coverage and short read lengths produced by next generation sequencing technologies the number of pairs that need to be compared can be huge. Since these comparisons need to be made many times, it is important that they be as efficient as possible [79]. Depending on the technology used the first step after sequencing may require an *all against all* comparison of the reads. Techniques for solving the all against all problem tend to focus on highly parallelisable, succinct data structures with a low memory footprint but not on the fastest execution time. We do not consider this problem in this chapter but a problem arising from the use of *paired-end* reads in shotgun sequencing. We focus on fast *pairwise* comparison of read overlaps to merge overlapping paired-end reads.

In paired-end sequencing a DNA fragment is sequenced in both directions rather than just one, the result is two short reads with an unknown sequence in the middle of approximately known length. Paired-end reads provide some of the benefits of longer read lengths without actually requiring the technology to sequence reads of that length. Paired-end reads can also simplify the detection of certain genetic variations by seeing how and where each paired read is aligned. When performing paired-end sequencing

it had previously been the case that the length of each read would be less than half the length of the DNA fragment being sequenced. Recent improvements to the Illumina family of sequencers has increased the read lengths such that they are now long enough that when a DNA fragment is sequenced the resulting reads will overlap rather than have an unknown middle section. This overlap means that it is possible to merge the two reads into a single long read of high quality. The chance that the sequencer has introduced an error increases exponentially the further down the read we consider, so merged paired end reads allow for far greater confidence in the latter half of the read than would normally be possible. So merging paired-end reads may be beneficial for reasons including, correcting sequencing errors, higher quality single reads and larger initial fragments for de-novo assembly. It is for these reasons that we efficient computation of this is the focus of this chapter.

Our Contribution. In this chapter we propose a new approach to the computation of approximate pairwise overlaps under Hamming and edit distance based on the computation of an approximate version of the prefix table. We define and provide algorithms for the computation of the prefix table under Hamming and edit distance. In addition we outline applications to other important problems such as approximate pattern matching. Showing that the simple algorithm performs well when compared to fast practical algorithms.

4.1 Efficient Computation of π_k^H and β_k^H

In order to provide an overview of our results and algorithms, we begin with a few definitions. The *border* of a string x is a prefix of x that is also a suffix of x. The *border array* $\beta = \beta[0\,..\,n-1]$ of x gives the length $\beta[i]$ of the longest border of every prefix $x[0\,..\,i]$, $0 \leq i < n$, of x. It is computed by an elegant algorithm in time $\Theta(n)$ [39, 122], and has the property that for every r^{th} longest border $\beta^r[i] > 0$, $\beta^{r+1}[i]$ is the length of the $(r+1)^{\text{th}}$ longest border, where β^r denotes r applications $\beta[\beta[\ldots\beta[i]\ldots]]$ of this function. Thus β specifies *all* the borders of every prefix of x. The *prefix table* $\pi = \pi[0\,..\,n-1]$ of x gives the length $\pi[i]$ of the longest substring beginning at position i, $0 \leq i < n$, of x, that equals a prefix of x. The prefix table was introduced in [90] to compute repetitions; it has since prominently appeared in [39, 123].

Observation 18. *If $x \equiv_k^H y$, then for every $i, j \in 0, \ldots, n-1$, $i \leq j$, $x[i\,..\,j] \equiv_k^H y[i\,..\,j]$.*

We can now define the *k-prefix table* π_k^H of x: for every i, $0 \leq i < n$,

$\pi_k^H[i] = \ell$ is the length of the longest prefix of x such that $x[i\,..\,i+\ell-1] \equiv_k^H x[0\,..\,\ell-1]$. By Observation 18, if $\pi_k^H[i] = \ell$, it follows that every prefix $x[0\,..\,j]$, $i \leq j \leq i+\ell-1$, has a k-border of length $j-i+1$. Thus, as for regular prefix tables, π_k^H determines all the k-borders of x [39]. Similarly we can define the *k-border* as a prefix of a string x at Hamming distance at most k from a suffix. The *k-border array* β_k^H can then be defined in the same way as the k-prefix table, but it should be noted that β_k^H specifies only the length of the *longest* border at each position i, not the lengths of shorter borders as its exact version would. This is a consequence of the nontransitivity under the distance model. Analogously, we can define the *k-prefix table* π_k^E of x and the *k-border array* β_k^E of x under edit distance.

In Section 4.1, we present two algorithms to compute π_k^H: a practical one requiring average-case time $\Theta(kn)$; and another requiring worst-case time $\Theta(kn)$; we then show how to compute β_k^H from π_k^H in time $\Theta(n)$.

In Sections 4.2 and 4.3, we show how the computation of π_k^H can be used to greatly speed up two computations of interest in computational biology and elsewhere. The first of these is approximate string matching with k-mismatches (see [39] for a definition) where given a text t of length n the problem is to search for occurrences of a pattern x of length $m < n$ at Hamming distance at most k from x. The original algorithms proposed for this problem [80, 53] require time $\mathcal{O}(kn)$. Shortly thereafter a $\mathcal{O}(\sqrt{m \log m} n)$-time algorithm was proposed [2], with a time requirement independent of k and asymptotically faster than its predecessors for $k \geq \sqrt{m \log m}$. About 13 years ago the asymptotically fastest algorithm was proposed, executing in time $\mathcal{O}(\sqrt{k \log k} n)$, as well as an alternative $\mathcal{O}((n + (nk^3)/m) \log k)$-time algorithm [7]. About 10 years ago an optimal average-case algorithm was proposed, executing in time $\mathcal{O}(n(k + \log_\sigma m)/m)$, only if $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$ [50]. Section 4.2 shows how to use π_k^H of xt to solve this problem in average-case time $\mathcal{O}\big(n + k(k+1)n/\sigma^{m/(k+1)}\big)$ — in practice, for moderate k, essentially linear in n. We also consider the well-known problem of computing the longest approximate overlap of two strings x of length m and y of length $n \geq m$ with k-mismatches. This overlap can be found in time $\mathcal{O}(kn)$ [79]. In Section 4.3, we present a very simple algorithm, based on the computation of β_k^H from π_k^H, that in time $\Theta(kn)$ not only solves the overlap problem for two strings, but also for every prefix of those strings.

Finally, in Section 4.4, we present an algorithm based on incremental string comparison techniques to compute π_k^E in worst-case time $\Theta(kn)$.

We present two algorithms that iteratively overwrite $\pi = \pi_0$ with π_j^H, $j \geq 1$, until $j = k$. The first requires average-case time $\Theta(kn)$ and the

second worst-case time $\Theta(kn)$. We then compute β_k^H from π_k^H in time $\Theta(n)$.

4.1.1 Average-case Algorithm for Computing π_k^H

The first algorithm is very simple and fast in practice. As we show below, it executes in average-case time $\Theta(n)$ for each of the k iterations. The fast runtime of the algorithm is based on the lemma below, establishing the expected number of symbol comparisons required to find a mismatch.

Fact 19. *The expected number of letter comparisons required for each i in algorithm k-PrefixTable-Simple is less than 3.*

Proof. On an alphabet of size σ, the probability that two random strings of length ℓ are equal is $(1/\sigma)^\ell$. Let $r = 1/\sigma$, there is probability r^ℓ the first ℓ symbols match. Thus the expected number of positions matched before inequality occurs is $S = r + 2r^2 + \cdots + (n-1)r^{n-1}$, for some $n \geq 2$. Hall & Knight [58, p. 44] tell us that $S = r(1-r^{n-1})/(1-r)^2 - (n-1)r^n/(1-r)$, which as $n \to \infty$ approaches $r/(1-r)^2 \leq 2$ for all r. Thus S, the expected number of matching positions for each i, is less than 2, and hence the expected number of letter comparisons required for each i in algorithm k-PrefixTable-Simple is less than 3. □

By Fact 19, we obtain the following.

Theorem 20. *Given a string x of length n, the prefix table π of x, and an integer threshold $k < n$, algorithm k-PrefixTable-Simple computes π_k^H in average-case time $\Theta(kn)$ and space $\Theta(n)$.*

4.1.2 Worst-case Algorithm for Computing π_k^H

Observation 21. *If $\pi_k^H[i] = \ell$, $0 \leq i < n$, then $x[0\,..\,\ell-1] \equiv_k^H x[i\,..\,i+\ell-1]$ and $x[\ell] \neq x[i+\ell]$.*

Computing the value of ℓ is equivalent to finding the longest common extension, denoted by lce, of the suffixes starting at $\pi_{j-1}^H[i]+1$ and $i+\pi_{j-1}^H[i]+1$, $1 \leq j \leq k$. To achieve $\Theta(n)$-time computation of each table we must be able to compute the lce of two suffixes in constant time.

Let SA denote the array of positions of the sorted suffixes of x, i.e. for all $1 \leq r < n$, we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. Let $\mathsf{lcp}(r,s)$

ALGORITHM k-PrefixTable-Simple(x, n, π, k)
{Where x is a string, n is the length of x and π is the prefix table of x. }
for $j \leftarrow 1$ to k **do**
Nothing to do for $i = 0$.
for $i \leftarrow 1$ to $n - 1$ **do**
$\delta \leftarrow i + \pi[i]$;
Nothing to do if $i + \pi[i] > n$.
if $\delta \leq n$ **then**
repeat
$\delta \leftarrow \delta + 1$;
until $\delta > n$ **or** $x[\delta - i] \neq x[\delta]$
end if
$\pi[i] \leftarrow \min(\delta - i, n - i)$;
end for
end for
return π;

denote the length of the longest common prefix of the strings $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$, for all $0 \leq r, s < n$, and 0 otherwise. Let LCP denote the array defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$, for all $1 < r < n$, and $\mathsf{LCP}[0] = 0$. We perform the following linear-time and linear-space preprocessing: (i) compute arrays SA and iSA of x [103]; (ii) compute array LCP of x [43]; and (iii) preprocess array LCP for range minimum queries, that we denote by $\mathsf{RMQ}_{\mathsf{LCP}}$ [44]. With the preprocessing complete, the lce of two suffixes of x starting at positions p and q can be computed in constant time in the following way (see [63] for details).

$$\mathsf{lce}(p, q) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[p] + 1, \mathsf{iSA}[q])]$$

Therefore, we obtain the following.

Theorem 22. *Given a string x of length n, the prefix table π of x, and an integer threshold $k < n$, algorithm k-PrefixTable computes π_k^H in worst-case time $\Theta(kn)$ and space $\Theta(n)$.*

4.1.3 Computing β_k^H from π_k^H

Here we give a $\Theta(n)$-time algorithm to compute β_k^H from π_k^H. Its computation is based on the following relationship between k-prefix tables and k-border arrays.

ALGORITHM k-PrefixTable(x, n, π, k)
{Where x is a string of length n, π is the prefix table of x and k is the allowed mismatches. }
Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of x.
for $j \leftarrow 1$ to k **do**
  Nothing to do for $i = 0$.
  for $i \leftarrow 1$ to $n-1$ **do**
    $\delta \leftarrow \pi[i] + 1 + \mathsf{lce}(\pi[i]+1, i+\pi[i]+1)$;
    $\pi[i] \leftarrow \min(\delta, n-i)$;
  end for
end for
return π;

Lemma 23. *Let ℓ be the largest index in β_k^H which has been correctly updated, and let i be the smallest index such that $i + \pi_k^H[i] > \ell$ and $i \leq \ell$. Then $\beta_k^H[i + \pi_k^H[i] - r - 1] = \pi_k^H[i] - r$, for all $0 \leq r < i + \pi_k^H[i] - 1 - \ell$.*

Proof. Recall that, by Observation 18, we know that if $\pi_k^H[i] > 0$ then the prefix of length $i + \pi_k^H[i] - j - 1$ has a k-border of length $\pi_k^H[i] - j$, for all $0 \leq j < \pi_k^H[i]$.

We update $\beta_k^H[i+\pi[i]-r-1]$, for all $0 \leq r < i+\pi_k^H[i]-1-\ell$, when we find some index $i \leq \ell$ such that $i+\pi_k^H[i] > \ell$. As $\ell < i+\pi_k^H[i]-r-1 < i+\pi_k^H[i]$, no $\beta_k^H[i + \pi_k^H[i] - r - 1]$ has been assigned a value, and for all subsequent j, such that $i < j < i + \pi_k^H[i]$ and $j + \pi_k^H[j] > \ell$, the k-borders given by j must be smaller than the k-borders given by i for the same prefix. Or more formally, $\pi_k^H[i] - r > \pi_k^H[j] - r'$, *iff* $i + \pi_k^H[i] - r = j + \pi_k^H[j] - r'$, for all $0 \leq r' < j+\pi_k^H[j]-1-\ell$. Therefore, the longest k-border is given by $\pi_k^H[i]-r$ and $\beta_k^H[i + \pi_k^H[i] - r - 1] = \pi_k^H[i] - r$, for all $0 \leq r < i + \pi_k^H[i] - 1 - \ell$. □

By Lemma 23, each index of β_k^H in algorithm k-BorderArray is updated only once and each index in π_k^H is read only once. So we have $2n$ operations in total. Therefore, we obtain the following.

Theorem 24. *Given array π_k^H of x of length n, algorithm k-BorderArray computes β_k^H in worst-case time and space $\Theta(n)$.*

All k-borders of x can be computed in time $\mathcal{O}(n^2)$ by the algorithm in [121]. We can compute all k-borders of x directly from π_k^H, thus in time $\Theta(kn)$: if $\pi_k^H[i] + i = n$, then $\pi_k^H[i]$ is the length of a k-border of x, for all $0 \leq i < n$.

ALGORITHM k-BorderArray(π_k, n)
{Where π_k is a prefix table of a string of length n.}
$\beta_k[0] \leftarrow 0$;
$\ell \leftarrow 0$;
for $i \leftarrow 1$ to $n-1$ **do**
Nothing to do if $i + \pi_k[i] - 1 < \ell$.
if $i + \pi_k[i] - 1 \geq \ell$ **then**
for $r \leftarrow 0$ to $i + \pi_k[i] - 1 - \ell$ **do**
$\beta_k[i + \pi_k[i] - r - 1] \leftarrow \pi_k[i] - r$;
end for
$\ell \leftarrow i + \pi_k[i]$;
end if
end for
return β_k

4.2 Application I: Approximate String Matching with k-mismatches via Filtering π_k^H

In this section, we present FPT, an algorithm for approximate string matching with k-mismatches. Algorithm FPT is based on Filtering the k-Prefix Table. Given a pattern x of length m, a text t of length $n > m$, and an integer threshold $k < m$, an outline of algorithm FPT is as follows.

1. Construct $T = xt$, and compute the prefix table π_0 of T.
2. The pattern x is split in $k+1$ fragments of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$.
3. Match the $k+1$ fragments against the text t using Aho Corasick automaton [41]. Let $\mathcal{L}$ be a list of tuples of size Occ, where $< \mathsf{id}, p > \in \mathcal{L}$ is a tuple such that $0 \leq \mathsf{id} \leq k$ is the fragment identifier, and $0 \leq p < n$ is the position that the fragment occurs in t.
4. Using these occurrences we could *invalidate* (filter out) the positions on π_0 that can never give a match if we extend them, i.e. we apply the partitioning technique [131]. Equivalently, for each tuple $< \mathsf{id}, p > \in \mathcal{L}$, we *validate* $\pi_0[m + p - \mathsf{id} \times \ell_{\mathsf{id}}]$, where ℓ_{id} is the length of the respective fragment.
5. Compute only the *valid* positions of π_i^H, for all $1 \leq i \leq k$, using algorithm k-PrefixTable.

6. If $\pi_k^H[i] \geq m$, x occurs at starting position $i-m$ of T, for all $m \leq i \leq n$.

Theorem 25. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, a text t of length $n > m$ drawn from Σ, and an integer threshold $k < m$, algorithm FPT requires average-case time $\mathcal{O}(n + k(k+1)n/\sigma^{m/(k+1)})$ and space $\mathcal{O}(n)$.*

Proof. The computation of the prefix table π_0 of $T = xt$ requires time and space $\mathcal{O}(n+m)$ (Step 1) [39]. Splitting the pattern x takes time $\mathcal{O}(k)$ (Step 2). The Aho-Corasick automaton of the $k+1$ fragments requires time $\mathcal{O}(m)$ with search time $\mathcal{O}(n + Occ)$ (Step 3) [41]. Validating positions of π_0 takes time $\mathcal{O}(Occ)$ (Step 4). Computing the valid positions of $\pi_1^H, \ldots, \pi_k^H$ requires time $\mathcal{O}(kOcc)$ (Step 5)—see Section 4.1.2. Reporting the output requires time $\mathcal{O}(n)$ (Step 6). Since the expected number Occ of occurrences of the $k+1$ fragments is $\mathcal{O}((k+1)n/\sigma^{m/(k+1)})$, algorithm FPT requires average-case time $\mathcal{O}(n + k(k+1)n/\sigma^{m/(k+1)})$. □

Corollary 26. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, a text t of length $n > m$ drawn from Σ, and an integer threshold $k = \mathcal{O}(m/\log m)$, algorithm FPT requires average-case time $\mathcal{O}(n)$.*

Proof. Algorithm FPT achieves average-case time $\mathcal{O}(n)$ *iff*

$$k(k+1)n/\sigma^{m/(k+1)} \leq cn$$

for some fixed constant c. Let $r = m/(k+1)$. We have $k(k+1)n/\sigma^r \leq cn$. Since $k < m$, we can (pessimistically) replace k by $m-1$. Then we have

$$m(m-1)n/\sigma^r \leq cn.$$

Solving for r, and using $k \leq m/r - 1$, gives the maximum value of k, that is $k = \mathcal{O}(m/\log m)$. □

By FPT-Simple, we denote the same algorithm apart from Step 5, where algorithm k-PrefixTable is replaced by algorithm k-PrefixTable-Simple. By applying Fact 19, it requires average-case time $\mathcal{O}(n)$, but because this approach avoids the computation of global data structures, it can be implemented in space $\mathcal{O}(m)$.

Corollary 27. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, a text t of length $n > m$ drawn from Σ, and an integer threshold $k = \mathcal{O}(m/\log m)$, algorithm FPT-Simple requires average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$.*

4.2.1 Experimental Results

We implemented FPT and FPT-Simple as library functions to perform approximate string matching with k-mismatches. They were implemented in the C programming language and developed under GNU/Linux operating system. Keeping in mind we wish to evaluate the practical efficiency of these two algorithms, we compared their performance to the respective performance of the following:

- Naive, an algorithm that considers all $\Theta(n)$ alignments of the text and the pattern, and counts mismatches at each alignment, stopping if more than k of them are found. This algorithm has worst-case time complexity $\mathcal{O}(mn)$, but average-case time complexity $\mathcal{O}(kn)$.
- Abrahamson, the algorithm presented in [2]. Even though this algorithm has worst-case time complexity $\mathcal{O}(\sqrt{m \log mn}n)$, we preferred it to the $\mathcal{O}(\sqrt{k \log kn}n)$-time algorithm presented in [7]. Both algorithms make extensive use of the Fast Fourier Transform to find the frequently occurring letters, however, the one proposed in [7] also requires the construction of the generalised suffix tree of x and t which is processed to allow constant-time *lowest common ancestor* queries, making it slower in practice. Due to this we opted to use the algorithm proposed in [2].
- FredNava, the algorithm with average-case optimal search time presented in [50]. The search-time complexity is $\mathcal{O}(n(k + \log_\sigma m)/m)$ and the space complexity is $\mathcal{O}(m^5\sigma^{\mathcal{O}(1)})$.

The experiments were conducted on a Desktop PC using 1 Intel Core Quad CPU Q9650 at 3.00GHz and 8GB of RAM and running under GNU/Linux. The implementation of algorithms FPT and FPT-Simple is distributed under the GNU General Public License (GPL) and is available at a website[1], which is set up for maintaining the source code. The implementations of algorithms Naive and Abrahamson were obtained from library StringPedia [124]; the implementation of algorithm FredNava was obtained via a personal communication with its author. Tables 4.1-4.3 illustrate elapsed-time comparisons for various pattern sizes and moderate values of k, using as text a corpus of English, protein, and DNA data taken from the Pizza&Chili website [108]. Different patterns were randomly picked from the text and the average elapsed time for each implementation with these patterns as input is presented.

[1] `http://www.inf.kcl.ac.uk/research/projects/asmf/`

Table 4.1: Elapsed-time and speed-up comparisons of algorithms Naive, Abrahamson, FPT, and FPT-Simple using English data ($\sigma = 128$) for $n =$ 50MB. *Algorithm FredNava was terminated by a segmentation fault

		Elapsed Time (s)					**Speed-up of FPT-Simple**			
m	k	Naive	Abrahamson	FredNava	FPT	FPT-Simple	Naive	Abrahamson	FredNava	FPT
2000	10	1.38	14.92	*	19.73	3.11	0.44	4.79	*	6.34
4000	25	2.54	26.28	*	20.15	3.48	0.72	7.55	*	5.79
8000	50	5.08	38.37	*	20.55	3.79	1.34	10.12	*	5.42
16000	100	9.67	52.32	*	20.86	4.17	2.31	12.54	*	5.00
32000	200	18.99	63.85	*	21.35	4.54	4.18	14.06	*	4.70
2000	25	2.93	14.90	*	20.50	3.73	0.78	3.99	*	5.49
4000	50	4.87	26.21	*	20.74	4.08	1.19	6.42	*	5.08
8000	100	9.70	38.62	*	20.98	4.20	2.30	9.19	*	4.99
16000	200	18.99	52.87	*	21.34	4.38	4.33	12.07	*	4.89
32000	400	37.40	64.64	*	22.12	4.54	8.23	14.23	*	4.87
2000	50	5.15	14.92	*	20.84	4.13	1.24	3.61	*	5.04
4000	100	9.28	26.59	*	20.96	4.18	2.22	6.36	*	5.01
8000	200	18.75	38.57	*	21.42	4.42	4.24	8.72	*	4.84
16000	400	37.13	52.48	*	22.37	4.50	8.25	11.66	*	4.97
32000	800	73.02	64.71	*	25.57	4.55	16.04	14.22	*	5.61

Table 4.2: Elapsed-time and speed-up comparisons of algorithms Naive, Abrahamson, FredNava, FPT, and FPT-Simple using protein data ($\sigma = 20$) for $n = 50$MB

		Elapsed Time (s)					**Speed-up of FPT-Simple**			
m	k	Naive	Abrahamson	FredNava	FPT	FPT-Simple	Naive	Abrahamson	FredNava	FPT
2000	10	1.11	19.98	13.34	22.32	2.74	0.41	7.29	4.87	8.15
4000	25	2.50	32.39	14.85	23.24	3.72	0.67	8.71	3.99	6.25
8000	50	4.61	57.20	14.92	24.12	4.16	1.11	13.75	3.59	5.80
16000	100	8.80	70.61	15.16	24.46	4.76	1.85	14.83	3.18	5.14
32000	200	17.20	81.77	15.16	24.73	4.97	3.46	16.45	3.05	4.98
2000	25	2.44	19.84	15.01	25.04	3.54	0.69	5.60	4.24	7.07
4000	50	4.55	32.00	14.97	23.72	4.24	1.07	7.55	3.53	5.59
8000	100	8.66	56.80	15.04	24.25	4.64	1.87	12.24	3.24	5.23
16000	200	17.21	70.71	15.18	24.88	4.86	3.54	14.55	3.12	5.12
32000	400	33.45	81.19	15.12	26.25	4.92	6.80	16.50	3.07	5.34
2000	50	4.67	19.88	14.93	23.88	4.17	1.12	4.77	3.58	5.73
4000	100	8.59	32.47	15.10	24.58	4.72	1.82	6.88	3.20	5.21
8000	200	17.18	56.93	15.00	25.16	4.78	3.59	11.91	3.14	5.26
16000	400	33.33	70.81	15.19	28.44	4.78	6.97	14.81	3.18	5.95
32000	800	66.76	80.90	15.22	36.03	5.08	13.14	15.93	3.00	7.09

Table 4.3: Elapsed-time and speed-up comparisons of algorithms Naive, Abrahamson, FredNava, FPT, and FPT-Simple using DNA data ($\sigma = 4$) for $n = 50$MB

		Elapsed Time (s)					Speed-up of FPT-Simple			
m	k	Naive	Abrahamson	FredNava	FPT	FPT-Simple	Naive	Abrahamson	FredNava	FPT
2000	10	3.14	14.88	3.36	22.71	3.36	0.93	4.42	1.00	6.75
4000	25	6.73	16.00	4.50	22.81	3.35	2.00	4.77	1.34	6.80
8000	50	12.74	16.69	4.32	22.96	3.48	3.66	4.79	1.24	6.59
16000	100	24.86	19.01	4.40	23.18	3.62	6.86	5.25	1.21	6.40
32000	200	49.28	20.38	4.40	23.19	3.86	12.76	5.27	1.13	6.00
2000	25	6.83	14.82	4.49	22.89	3.36	2.03	4.41	1.33	6.81
4000	50	12.82	15.83	4.28	22.91	3.43	3.73	4.61	1.14	6.67
8000	100	24.78	16.72	4.31	22.94	3.50	7.08	4.77	1.23	6.55
16000	200	49.17	19.01	4.47	23.15	3.64	13.50	5.22	1.22	5.16
32000	400	98.23	20.29	4.40	23.31	3.88	25.31	5.22	1.21	6.00
2000	50	12.89	14.86	4.31	23.25	3.42	3.76	4.34	1.26	6.79
4000	100	25.05	15.65	4.31	24.02	3.50	7.15	4.47	1.23	6.86
8000	200	48.90	18.98	4.31	25.30	3.68	13.28	5.15	1.17	6.87
16000	400	97.55	19.04	4.40	26.06	3.78	25.80	5.03	1.16	6.89
32000	800	195.18	20.26	4.40	27.53	4.10	47.60	4.94	1.07	5.55

As demonstrated by the experimental results, algorithm FPT-Simple is in most cases the fastest. Algorithm Naive is the fastest for small m and k. Algorithm FredNava with English data was terminated by a segmentation fault during preprocessing stage due to lack of memory. Algorithms FredNava and FPT-Simple with DNA data perform very similarly. Another observation, also suggested by Corollaries 26 and 27, is that the FPT-based algorithms are essentially *independent* of m for moderate values of k.

4.3 Application II: Longest Approximate Overlap of Two Strings with k-mismatches

Finding approximate overlaps is the first phase of many sequence assembly methods. Given a set of r strings and an error rate ϵ, the goal is to find, for all pairs of strings, their suffix/prefix matches (overlaps) that are within edit or Hamming distance $k = \lceil \epsilon \ell \rceil$, where ℓ is the length of the overlap. Many existing solutions focus on applications where r is large, the average string length is small, and k is small; and therefore make use of techniques such as *backward backtracking* and/or *suffix filters* to save space [127]. However,

algorithms are also needed to merge overlapping paired-end reads, in the case when $r = 2$, while correcting mismatches and uncalled bases [136]. Here we focus on the case where $r = 2$, although our algorithm can be used to compute the approximate overlap between r strings in time $\Theta(r^2Nk)$, where N is the average length of the r strings.

Given a string x of length m, a string y of length $n \geq m$, and an integer threshold $k < m$, this overlap under edit or Hamming distance can be found in time $\mathcal{O}(kn)$ by the algorithm of [79]. Here, we propose a simple alternative algorithm, for Hamming distance, that requires time $\Theta(kn)$ and space $\Theta(n)$. Furthermore, notice that the proposed algorithm not only computes the longest approximate overlap of x and y with k-mismatches, but also of all their prefixes.

1. Construct $T = yx$, and compute the prefix table π_0 of T.
2. Compute the arrays $\pi_1^H, \ldots, \pi_k^H$ of T using algorithm k-PrefixTable.
3. Compute the arrays $\beta_0^H, \beta_1^H, \ldots, \beta_k^H$ of T using algorithm k-BorderArray.
4. $\beta_0^H[m+n-1], \beta_1^H[m+n-1], \ldots, \beta_k^H[m+n-1]$ give the longest approximate overlap of x and y with $0, 1, \ldots, k$ mismatches, respectively.

As mentioned above, existing solutions for the overlap problem consider very different sets of parameters, and so are not directly comparable with ours. Similar to Section 4.2, we anticipate that using algorithm k-PrefixTable-Simple to compute the k-prefix table and, then, algorithm k-BorderArray to compute the k-border array would yield a very fast and simple solution.

4.4 Efficient Computation of π_k^E and β_k^E

In this section, we consider the prefix table under edit distance and present an efficient algorithm for its computation. The computation is heavily based on incremental string comparison techniques so first we give an overview of these techniques. The incremental string comparison problem was introduced by Landau *et al.* in [79]. The authors considered the following problem: given the edit distance between two strings A and B, how can the edit distance between A and bB; or Bb be efficiently derived, where b is an additional letter? Given a threshold on the number of differences k, they solve this problem and allow prepending and appending of letters in time $\mathcal{O}(k)$ per operation. Later in [61] a generalisation of the problem was considered where

prefixes can be deleted and prepended to A or B with time complexity of $\mathcal{O}(k)$ per letter.

The idea in both [79] and [61] is the efficient computation of *h-waves*. In the standard dynamic programming matrix, we say that a cell $\mathsf{D}[i,j]$ is on the diagonal d *iff* $j-i=d$. For each diagonal, we may have a lowest cell with value h; if $\mathsf{D}[i,j]=h$ and $\mathsf{D}[i+1,j+1]=h+1$ then $\mathsf{D}[i,j]$ is this cell for diagonal $j-i$. The h-wave, for all $0 \leq h \leq k$, is the position of all these cells across all diagonals, that is, a list H_h of length $\mathcal{O}(k)$, where each entry is a pair (i,j) such that $\mathsf{D}[i,j]=h$ and $\mathsf{D}[i+1,j+1]=h+1$. Note that the i-th wave can only contain entries on diagonal zero and the i diagonals either side of it, so for $0 \leq i \leq k$ every wave has size $\mathcal{O}(k)$. These h-waves define the entire dynamic programming matrix due to monotonicity properties. For any diagonal d, if we know the position of the lowest cell on d with value h and $h+1$, then we also know the value of every cell between these two cells: it must be $h+1$. So given the h-waves of the matrix, for all $0 \leq h \leq k$, we have all the information from the standard dynamic programming matrix. The key result from our perspective is the following. Let $\mathsf{cat}(u',u)$ denote the string obtained by concatenating string u' and string u. Let $\mathsf{del}(\alpha,u)$ denote the string obtained by deleting the prefix of length α from string u. Further let D' denote the standard dynamic programming matrix of $\mathsf{cat}(\mathsf{A}',\mathsf{A})$ and $\mathsf{del}(t_2,\mathsf{B})$, where $|\mathsf{A}'|=t_1$.

Theorem 28 ([61])**.** *The* 0*-wave,* 1*-wave, ... , and* k*-wave of matrix* D' *can be computed in time* $\mathcal{O}((t_1+t_2)k)$.

Let $\mathsf{DP}(x,y,k)$ denote the dynamic programming algorithm for computing the edit distance (at most k) between strings x and y. This algorithm requires time $\Theta(kn)$ [126]. Let D denote the resulting dynamic programming matrix of size $\Theta(kn)$. Further, let $\mathsf{GH}(\mathsf{D})$ denote the function to extract $\mathsf{H}_{0,\ldots,k}$ from D, and let $\mathsf{ISC}(\mathsf{H}_{0,\ldots,k},x,y,\alpha)$ denote the incremental string comparison function that updates $\mathsf{H}_{0,\ldots,k}$ for x and $\mathsf{del}(\alpha,y)$. We are now in a position to outline the computation of the prefix table under edit distance. For each position i, for all $1 \leq i < n$, we compute $\mathsf{H}_{0,\ldots,k}$ for x and $x[i\,..\,n-1]$. We then check the k-wave of the dynamic programming matrix to find the length ℓ of the longest prefix of x such that $x[i\,..\,i+j-1] \equiv_k^E x[0\,..\,\ell-1]$ for $\ell \geq k$ and $\ell-k \leq j \leq \ell+k$.

The k-wave is stored as a linked list of size $2k$ that specifies for each diagonal the lowest cell with value k. To find this longest prefix, we simply iterate through the linked list of the k-wave and keep track of the diagonal δ with the lowest cell on the k-wave. If a diagonal has no cell with value k then clearly that diagonal has reached the last row of the dynamic programming

ALGORITHM k-PrefixTable-ED(x, n, k)
{Where x is a string of length n and k is the allowed differences.}
$\pi_k^E[0] \leftarrow n$;
$\mathsf{D} \leftarrow \mathsf{DP}(x, x[1\,..\,n-1], k)$; $\mathsf{H}_{0,\ldots,k} \leftarrow \mathsf{GH}(\mathsf{D})$;
for $i \in \{1, n-1\}$ **do**
  $\ell \leftarrow -1$;
  for $(u, v) \in \mathsf{H}_k$ **do**
    if $v > u$ **then**
      $w \leftarrow u$;
    else
      $w \leftarrow v$;
    end if
    if $w \geq \ell$ **then**
      $\ell \leftarrow w$; $\delta \leftarrow v - u$;
    end if
  end for
  if $\delta > 0$ **then**
    $\pi_k^E[i] \leftarrow \ell$;
  else
    $\pi_k^E[i] \leftarrow \ell - \delta$;
  end if
  if $i < n - 1$ **then**
    $\mathsf{H}_{0,\ldots,k} \leftarrow \mathsf{ISC}(\mathsf{H}_{0,\ldots,k}, x, x[i\,..\,n-1], 1)$;
  end if
end for
return π_k^E;

matrix. This procedure can be seen in algorithm k-PrefixTable-ED. Hence we obtain the following.

Theorem 29. *Given a string x of length n and an integer threshold $k < n$, algorithm k-PrefixTable-ED computes π_k^E in worst-case time and space $\Theta(kn)$.*

The conversion between π_k^E and β_k^E is performed in exactly the same way as for Hamming distance (algorithm k-BorderArray).

4.5 Conclusions and Future Work

Here we presented theoretically and practically efficient algorithms to compute the k-prefix table and the k-border array of a string under both Hamming and edit distance. It was then shown how these data structures can be used for efficient approximate string matching under Hamming distance; and for the important problem of computing approximate pairwise overlaps. We performed computational experiments for approximate string matching with k-mismatches, which demonstrate that our algorithms are competitive for moderate values of k — which is usually the case in real-life applications.

5

Computing Repetative Features in Weighted Strings

Strings are fundamental for representing data in bioinformatics, from DNA/RNA reads to mass spectrometry data. With the increase in data produced by NGS technologies, highly efficient algorithms for biological problems have become even more important and research into them has increased in recent years. Although the data are essentially just strings, it is still important to question what is the best representation of the data. All data sets come with their own particular challenges for processing, but biological data have some specific and interesting characteristics which can be exploited for efficient algorithmic solutions and improved analysis. There may be a certain amount of ambiguity in a DNA sequence which can be exploited in their analysis.

DNA sequences derived from next generation sequencers may have errors and uncertainty introduced to them by the sequencing technologies. In some situations the uncertainty or errors can be modelled as a wildcard character, but depending on the source and severity of the uncertainty it may be possible to express it more subtly. It may not be possible to give the exact base that occurs at some position, but it may be possible to give the probability that each base occurs. If this is possible then the most natural representation of this data is as a *weighted string*. A *weighted string* is a string where each position contains the entire alphabet and the probability that each letter may occur. Representation as a weighted string may allow for a more accurate analysis of data of this nature. In this chapter we consider a number of problems related to the analysis of data represented as weighted strings.

The problems considered in this chapter are as follows:

Problem 30 (Inverted Repeats in Weighted Strings)**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all maximal inverted repeats in x.*

Problem 31 (Inverted Repeats in Weighted Strings with k-mismatches)**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0,1]$, find all maximal inverted repeats with up to k-mismatches in x.*

Problem 32 (Tandem Repeats in Weighted Strings)**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0,1]$, find all repetitions in x.*

Problem 33 (Covers in Weighted Strings)**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0,1]$, find all covers of x.*

Our Contribution. In this chapter, we consider the computation of a number of repetitive structures in weighted strings. We first consider the computation of exact and approximate inverted repeats and give $\mathcal{O}(n)$ and $\mathcal{O}(kn)$ algorithms for their computation. We then present algorithms for computing all tandem repeats and all covers in a weighted sequence. We improve on the time complexity of the best-known algorithm for these problems in weighted strings from time $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. This time complexity is optimal for the case of tandem repeats.

5.1 Colouring a Weighted String

The first stage of all the algorithms presented in this chapter is to perform a simple filtering scheme on the weighted string to filter out all those letters that are below the threshold. This is required as if the alphabet is not constant, we may have many letters with low occurrence probability that are not of interest. We simply read the entire string and keep only those letters with probability greater than or equal to $1/z$; these are at most z for each position, so still constant. We are thus left with a string of size $\mathcal{O}(n)$, and the entire stage takes time $\mathcal{O}(\sigma n)$. For clarity of presentation, in the rest of this chapter, we assume that the string resulting from this filtering step is the input weighted string x.

After this filtering stage, we perform a colouring stage on x, similar to the one performed in [64], which assigns a colour to every position in x according to the following scheme:

- mark position i *black* (`B`), if *none* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$;
- mark position i *grey* (`G`), if *one* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$;

Position	0	1	2	3	4	5	6	7	8	9	10
x	A	(A, 0.1) (C, 0.8) (G, 0.1) (T, 0.0)	T	T	(A, 0.5) (C, 0.5) (G, 0.0) (T, 0.0)	T	C	(A, 0.6) (C, 0.2) (G, 0.0) (T, 0.2)	T	T	T
Colour	W	G	W	W	B	W	W	G	W	W	W

Table 5.1: Colouring of x for $1/z = 1/2$.

- mark position i *white* (W), if *one* of the possible letters at position i has probability of occurrence 1.

An example of the colouring scheme can be seen in Table 5.1.

From here we additionally assume that $z \geq 2$ as for $z < 2$ all positions are either white or grey.

After the colouring stage, we perform a generation stage, similar to the one performed in [64], where a set of factors of x is generated; we refer to this set as *extended factors*. The procedure outlined below is similar to the generation step presented above, however, here, we generalise the procedure to any finite alphabet, by noting that the branching factor for any finite alphabet can be no more than $1/(1/z) + 1 = z + 1$ for the first branching and z for the rest; as no more than z letters can have a probability greater than or equal to $1/z$.

The generation of extended factors is performed once from each black position. We scan x from left to right and for the currently considered black position, we create a list of possible extended factors starting from this position. We generate a factor starting with each letter at the black position and one empty string. These will then be extended to create the extended factors starting from that black position. For each extended factor, the cumulative probability is maintained during its generation, and when its cumulative probability breaks the threshold we stop extending it. This probability is updated by considering the actual probability of occurrence for letters at black positions, but letters at grey positions are treated as letters at white positions (only one possible choice). Extending these factors is performed by continuing to scan x and appending to the currently considered factors the same single letter if the position is white or grey and by creating new factors at black positions. At a black position we copy each current extended factor and append one letter from the black position to each copy. We stop extending an extended factor when we reach a black position which causes

it to violate the threshold.

Example 34. *Let* $x = \mathtt{aab}[(\mathtt{a}, 0.5)(\mathtt{b}, 0.5)][(\mathtt{a}, 0.5)(\mathtt{b}, 0.5)]\mathtt{bab}$ *and* $1/z = 1/2$*. The colouring corresponding to* x *is* `WWWBBWWW`*. We scan* x *from left to right and extend until the first black position. At this point we have factor* `aab` *and at the black position we branch and get* `aaba` *and* `aabb`*. Extending these any further will violate the threshold so we stop. The extended factors generated from position* 0 *of* x *are as follows:* `aaba` *and* `aabb`*. We continue with factors* `a`*,* `b`*, and* ε *at the black position* 3*. We then reach the black position* 4 *and for factors* `a` *and* `b` *this violates the threshold so we stop. However, the empty string can be extended, so we split it into two strings, as we are at a black position, and continue extending it. The extended factors generated from the black position* 3 *are as follows:* `a`*,* `b`*,* `abab`*, and* `bbab`*. We continue with factors* `a`*,* `b`*, and* ε *at the black position* 4*. The extended factors generated from the black position* 4 *are as follows:* `abab`*,* `bbab`*, and* `bab`*.*

We recall an important lemma on how many black positions may be contained within any valid (or extended) factor of a weighted string. We also give a slightly modified proof for any finite alphabet.

Lemma 35 ([64])**.** *Given a weighted string* x *and a cumulative weight threshold* $1/z \in (0, 1]$*, any valid factor of* x *contains at most* $\lceil \log z / \log(\frac{z}{z-1}) \rceil$ *black positions.*

Proof. Consider a valid factor u of x containing ℓ black positions and no grey positions. Any letter at a black position has occurrence probability at most $1 - 1/z$. The cumulative occurrence probability of u with ℓ black positions is no more than $(1 - 1/z)^\ell$, and it must be the case that $(1 - 1/z)^\ell \geq 1/z$ since u is valid; by rearranging and taking logarithms we obtain the claimed result. □

From the generation of extended factors we get the following.

Lemma 36 ([64])**.** *A valid factor of* x *occurs in at least one of the extended factors of* x*.*

The sum of lengths of the extended factors is linear in n by Lemma 38. We give an alternative proof to the one given in [64] for any finite alphabet, both for completeness and as we improve the bounds on the constants slightly. For this alternative proof, we first need to show the following lemma.

Lemma 37. *Given a weighted string x and a cumulative weight threshold $1/z \in (0, 1]$, any valid factor of x occurs in at most $z^\ell(2\ell + 1)$ extended factors of x, where $\ell = \lceil \log z / \log(\frac{z}{z-1}) \rceil$.*

Proof. By the definition of extended factors, each black position initially generates $z + 1$ extended factors; z including the current black position and one that does not. At each subsequent black position, each extended factor may branch at most z times, and this occurs no more than ℓ times. From this we get that a black position generates no more than $z^\ell + z^\ell$ extended factors; z^ℓ from those that initially include the current black position and another z^ℓ from the one that does not.

Now consider some position i in the weighted string x. Position i can only be in extended factors generated by black positions to the left of it or at position i itself; and we know that an extended factor can contain at most ℓ black positions. Position i can only be contained in extended factors generated from the $\ell+1$ black positions to the left. For the first ℓ to the left, it can be contained in any extended factor but for the $\ell + 1$th black position to the left, it can only be contained in those extended factors which do not include the $\ell + 1$th black position.

By Lemma 36, all valid factors occur in extended factors and no valid factor can occur in strictly more extended factors than its respective single-letter valid factors. Therefore it is sufficient to determine, for some position i, the maximum number of occurrences of its single-letter valid factors in extended factors. From the above analysis, we can see that each position can be in at most $2\ell z^\ell + z^\ell = z^\ell(2\ell + 1)$ extended factors. □

We are now ready to establish the sum of lengths of extended factors.

Lemma 38 ([64])**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, the sum of lengths of the extended factors of x is $\mathcal{O}(n)$.*

Proof. Following the proof of Lemma 37, we see that each position is in no more than $z^\ell(2\ell + 1)$ extended factors where $\ell = \lceil \log z / \log(\frac{z}{z-1}) \rceil$. To establish the sum of lengths of extended factors it is sufficient to count how many extended factors each position is in; therefore the sum of lengths of all extended factors is no more than $z^\ell(2\ell + 1)n = \mathcal{O}(n)$. □

Finally we show a property of valid factors important for some of the later techniques.

Lemma 39. *A valid factor of x is in $\mathcal{O}(1)$ extended factors of x.*

Proof. Consider some position i in the weighted string. Position i can only be in extended factors generated by black positions to the left of it or at position i itself and we know that an extended factor can contain at most $\lceil \log z / \log(\frac{z}{z-1}) \rceil$ black positions. Clearly i can only be contained in extended factors generated from the $\lceil \log z / \log(\frac{z}{z-1}) \rceil$ black positions to the left. Each black position only generates $\mathcal{O}(1)$ extended factors and this proves the claim. □

These are some of the basic facts required in all of the algorithms presented in this chapter. Other techniques required will be presented in the process of developing our algorithms and techniques. First we consider the computation of exact and approximate inverted repeats.

5.2 Inverted Repeats in Weighted Strings

Inverted repeats or *hairpins* are factors of the string that occur with a reverse and complemented version of it in close proximity, where the complement is defined by the Watson-Crick base pairings; for example `ACTAGT` is an inverted repeat in the string `TTACTAGTTT`. Inverted repeats represent areas where there could exist complementary base pairings. In addition to representing areas of complementary base pairings inverted repeats can define the boundaries in transposons. These two properties play an important role in genome instability and contribute not only to cellular evolution and genetic diversity but also to mutation and disease.

A related biological feature is a *pseudoknot*; pseudoknots are a structural element known to play an important part in certain cell activity [30], as well as in viral infiltration [109]. Pseudoknots occur when two or more inverted repeats *overlap*, in the sense that one inverted repeat forms on part of an existing inverted repeat. Although there exists polynomial time algorithms for computing *some* pseudoknots, the problem is in general known to be *NP-Hard*. A number of heuristics have been proposed for this problem, the majority of which are based on dynamic programming [3] and struggle to detect the complex interactions of the non-nested base pairs that form pseudoknots. New methods based on stochastic context free grammars [27, 73] still struggle to detect the complex interactions involved. As inverted repeats are the building blocks for pseudoknots, efficient algorithms for the detection of inverted repeats may lead to more efficient heuristics for detecting pseudoknots.

There are a number of techniques which can be used to compute inverted

repeats in regular strings, but the case of weighted strings is until now unexplored. From a biological perspective, existing work on detecting inverted repeats tends to focus on probabilistic methods for their detection. One such method is that proposed in [72] which makes use of local alignments and probabilistic techniques. A well known statistical tool is the *Inverted Repeats Finder*[1] that uses a stochastic model of repeats to analyse DNA. The *Inverted Repeats Finder* is a popular tool which gives good results, however, none of the approaches mentioned above provide an exact solution to the problem of finding inverted repeats.

A related problem to the computation of inverted repeats is the detection of palindromes in strings. Within the algorithmic community there has been a great deal of work on the detection of palindromes in regular strings. The exact case of palindrome detection has been extensively studied with algorithms for a number of variations of the problem being proposed such as, finding the longest palindrome [91], computing maximal palindromes [57] and computing distinct palindromes [56]. The detection of approximate palindromes has also been considered, in [110] an algorithm for the computation of maximal palindromes under edit distance in time $\mathcal{O}(k^2n)$ was proposed; later improved in [61] were an $\mathcal{O}(kn)$ algorithm was presented. Although some of these techniques can be modified to compute inverted repeats, all the results mentioned above are only for regular strings and do not consider weighted strings.

In the following section we develop algorithms for computing inverted repeats in weighted strings. We consider the computation of exact and approximate inverted repeats in weighted strings and present $\mathcal{O}(n)$ and $\mathcal{O}(kn)$ algorithms respectively. Here we consider the case where the approximation measure is the Hamming distance at most k.

5.3 Computing Inverted Repeats

In this section we first introduce a few definitions and then, as a warm up, outline the basic approach we take and how it can be applied to finding exact and approximate inverted repeats under Hamming distance in regular strings. After this we present the main focus of this section and outline the modifications to deal with weighted strings for both the exact and k-mismatches cases.

In this section we focus only on the case of even inverted repeats; odd

[1]Inverted Repeats Finder can be found here http://tandem.bu.edu/irf/irf.download.html

inverted repeats can be handled with simple modifications to the approaches presented in this section. We begin by recalling the definition of an inverted repeats with respect to it's *centre*, an example of this can be seen in Example 41.

Definition 40. *A factor $w = u\bar{u}^R$ is an inverted repeat of radius $|u|$, centered around i if and only if the start position of $\bar{u}^R$ is $i + 1$.*

The approach taken in this chapter is to compute inverted repeats by their centres. We additionally define pairs of positions in an inverted repeat. For an inverted repeat $u\bar{u}^R$ at centre i of radius $|u|$ for $i - |u| + 1 \leq j \leq i$ we define the position $2i - j + 1$ as it's *corresponding position* in the inverted repeat and vice versa.

Example 41. *Given the string $x =$ `GAGAGAACCGT ACGGT`, then there exists a maximal inverted repeat (underlined in x), `ACCGTACGGT` of length 10, centred at position 10. There are also inverted repeats of length $2, 4, 6$ and 8 also centred at position 10 however they are not maximal.*

Given a string x and its reverse complement $\bar{x}^R$, the maximal inverted repeat centred at some position $1 < i < n - 1$ is the *Longest Common Extension* (lce) between $x[i+1]$ and $\bar{x}^R[n-i]$. To report all of these in linear time we must be able to compute $\mathsf{lce}(x[i+1], \bar{x}^R[n-i])$ in constant time. To do this we can compute the generalised suffix tree of x and x^R and process the tree for *Lowest Common Ancestor* (lca) queries. The lca of two suffixes in the suffix tree corresponds to the lce of the two suffixes. To compute the inverted repeats we proceed with the following scheme:

- Build the generalised suffix tree of x and $\bar{x}^R$;
- Process the suffix tree for lca queries;
- Find the lce for the suffixes starting at $(x[i + 1], \bar{x}^R[n - i - 1])$, for $0 < i < n - 1$;
- The result, for $0 < i < n - 1$, will be the maximal inverted repeat centred at $x[i]$.

The above scheme allows us to compute exact inverted repeats in regular strings in $\mathcal{O}(n)$ time.

5.3.1 Inverted Repeats with k-mismatches

To compute the maximal inverted repeats with up to k-mismatches the same processing as for exact inverted repeats is performed but the latter half of the algorithm is modified. The approach we adopt is similar to that used in [21] and consists of performing $k+1$ lce queries instead of a single lce query for each index. Assume we have performed one lce query and a mismatch occurs between index p and q, we now simply take another lce query from the suffixes starting at $p+1, q+1$. After performing this $k+1$ times we find the maximal inverted repeats with at most k-mismatches.

5.3.2 Extensions to Weighted Strings

For weighted strings the general approach of our algorithm is the same but the details are more complex. The most significant difference is that we must now account for ambiguous positions in the string. This ambiguity can be seen in the example below, at position 4 two characters have a probability above the threshold so either could be lead to valid factors. In this algorithm we make use of the weighted suffix tree of [64].

Example 42. *Let the following weighted string x and the cumulative weight threshold $1/z = 1/4$.*

Position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
x	A	C	T	T	(A, 0.5)	T	C	(A, 0.6)	T	T	T
					(C, 0.5)			(C, 0.2)			
					(G, 0.0)			(G, 0.0)			
					(T, 0.0)			(T, 0.2)			

First we give a brief introduction to the weighted suffix tree and then outline our algorithm.

The Weighted Suffix Tree

In [64], Iliopoulos *et al.* describe a method for constructing a *Weighted Suffix Tree* (WST) in linear time, for a constant cumulative probability threshold $1/z$, where z is a given constant. The WST stores the set of factors of a weighted string with probability of appearance greater than $1/z$. The WST is distinct from the probabilistic suffix tree in that is does not model any stochastic process and is designed to work in the same way as a suffix tree, meaning that it maintains optimal search times; something not possible

with the probabilistic suffix tree [113]. As the WST is designed to behave exactly as a suffix tree and it does not actually contain any information about the probability distribution of the weighted string within the suffix tree itself, instead it only represents those factors which have probability of occurring $\geq 1/z$. We give an informal definition of the structure as follows: Let x be a weighted string, for every suffix starting at position i we define a list of possible weighted factors so that the probability of appearance for each of them is greater than $1/z$. We define $\mathsf{WST}(x)$ the weighted suffix tree of a weighted string x, as the compressed trie of all the valid factors occurring in x.

The resulting suffix tree only consists of those factors which have probability greater than threshold $1/z$. We can use existing algorithms to process the weighted suffix tree for *Lowest Common Ancestor* (lca) queries, allowing us to compute the lce of two valid factors in constant time.

Exact Weighted Inverted Repeats

The approach presented in the previous section was sufficient for the exact non-weighted case, however, when considering weighted strings we now have some extra issues to take into account. Directly applying the approach outlined for regular strings may actually find inverted repeats of the form $u' = u\bar{u}^R$ where u and $\bar{u}^R$ are valid factors of probability greater than $1/z$. The problem is that the probability of u' occurring might be as low as $1/z^2$. To help overcome this problem we construct an additional data structure to efficiently report the maximum radius at any centre; this data structure takes $\mathcal{O}(n)$ time to build and occupies $\mathcal{O}(n)$ space.

Before we begin we first draw attention to the observation that black positions are the only positions which are ambiguous, although grey positions contain multiple characters only one can lead to a valid factor. Recall that a valid factor must have occurrence probability $\geq 1/z$, and that by Lemma 35 the number of black positions in a valid factor is constant. From this we can say that for a valid factor starting at position i of length m is uniquely defined by the characters in the black positions. For any other valid factor of length m starting at position i, all characters other than black positions must be the same. This leads us to the following conclusion about the probability of a factor in a weighted string:

Lemma 43. *Given a weighted string x of length n and a factor starting at position i of length m, its probability is uniquely defined by the characters in the black positions.*

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	A	C	T	T	(A, 0.5) (C, 0.5) (G, 0.0) (T, 0.0)	T	C	(A, 0.5) (C, 0.0) (G, 0.0) (T, 0.5)	T	T	T
BL$[i]$	∞	∞	∞	∞	0	4	4	0	7	7	7
BR$[i]$	4	4	4	4	0	7	7	0	∞	∞	∞

Table 5.2: Weighted string x with BR and BL array.

The above lemma tells us that for any centre its maximal radius is determined by the characters chosen at black positions and the radius may be different depending which characters are chosen. The above lemma also tells us that for any centre, all valid maximal inverted repeats at that centre can only differ by their radius and at most a constant number of positions surrounding the centre. In particular, by Lemma 35 they cannot differ by more than $\log z / \log(\frac{z}{z-1})$ positions to the left of the centre and no more than $\log z / \log(\frac{z}{z-1})$ positions to the right.

Although the general scheme of the algorithm remains the same for the weighted case, it must be considered that there may be many possible valid factors starting at any position in a weighted string. This means that instead of performing a single lce query it seems we may have to perform many. The approach taken is to initially compute the maximal inverted repeat at some centre without explicit consideration of the probabilities involved. We only consider the number of black positions it passes through, but not any affect that grey positions might have, or the actual probabilities at black positions. For each centre, we first compute the maximal inverted repeat with at most $\log z / \log(\frac{z}{z-1})$ black positions. Once the maximal inverted repeat with at most $\log z / \log(\frac{z}{z-1})$ black positions is found it will be shortened to the maximal valid inverted repeat. Before we outline our technique we compute the array BR which for a given position returns the position of the closest black position to it's right and array BL which does the same to the left. These arrays can be seen in Table 5.2 and can be computed in $\mathcal{O}(n)$ time along with the weighted suffix tree. We now focus on how to compute the maximal inverted repeat at a particular centre and show that this can be done in constant time per centre.

Consider a left pointer $\mathcal{L}$ at position $j \leq n-1$ and a right pointer $\mathcal{R}$ at some $i \geq j$. Let $\ell = \log z / \log(\frac{z}{z-1})$, $(x_{\mathcal{L}_0}, x_{\mathcal{L}_1}, .., x_{\mathcal{L}_{\ell-1}})$ be the sequence of positions of the closest ℓ black positions at or to the left of i in descending order of position and $(x_{\mathcal{R}_0}, x_{\mathcal{R}_1}, .., x_{\mathcal{R}_{\ell-1}})$ be the sequence of positions of

the closest ℓ black positions to the right of i in ascending order of position. These positions can be computed in constant time with the BL and BR arrays. Consider moving the pointers away from each other one position at a time comparing $x[\mathcal{L}]$ and $x[\mathcal{R}]$ under the complement relation, at some point one of the following will hold:

- $x[\mathcal{L}] \bar{\neq} x[\mathcal{R}]$.
- Either $x[\mathcal{L}]$, $x[\mathcal{R}]$ or both are black positions.
- $\mathcal{L} = 0$ or $\mathcal{R} = n - 1$.

For some centre j, we must be able to determine which of the above occurs and at what position for the following cases: $\mathcal{L}$ and $\mathcal{R}$ both at j; $\mathcal{L}$ on a black position to the left of j and $\mathcal{R}$ on it's corresponding position; $\mathcal{R}$ on a black position to the right of j and $\mathcal{L}$ on it's corresponding position. This corresponds to knowing, for each black position in $(x_{\mathcal{L}_0}, x_{\mathcal{L}_1}, .. x_{\mathcal{L}_{\ell-1}})$ and $(x_{\mathcal{R}_0}, x_{\mathcal{R}_1}, .. x_{\mathcal{R}_{\ell-1}})$, how far the radius of an inverted repeat centred at j can be extended before one of the previously mentioned cases occur. An important observation is that the above cases never cross a black position, so the positions considered are not ambiguous. First compute the following, where any factor starting at the specified positions can be chosen:

- Compute $\mathsf{lce}(x[j+1], \bar{x}^R[n-j-1])$.
- Compute $\mathsf{lce}(x[2j - x_{\mathcal{L}_r} + 2], \bar{x}^R[n - x_{\mathcal{L}_r} - 2])$, for $0 \leq r < \ell$.
- Compute $\mathsf{lce}(x[x_{\mathcal{R}_r} + 1], \bar{x}^R[n - x_{\mathcal{R}_r} + 2])$, for $0 \leq r < \ell$.

Although there may be multiple suffixes starting from each position any may be chosen as we only wish to compute the lce between the sections of consecutive white and grey positions, which are common to all factors starting at these positions. If the lce extends past the closest black position to the left of $\mathcal{L}$ or right of $\mathcal{R}$, we will shorten it to which ever position is closest. This can easily be computed as we know the position of the next black position in both directions. By Lemma 35 there are only a constant number of black positions in each list and therefore this entire step takes constant time for each position. The information computed above tells us, for each black position, if the radius of an inverted repeat centred at j can be extended until either $\mathcal{L}$ or $\mathcal{R}$ reaches a black position and if not, how far it can be extended before a mismatch occurs.

Determining the maximal inverted repeat with at most ℓ black positions centred at j can be computed by placing $\mathcal{L}$ and $\mathcal{R}$ at j; determining $(x_{\mathcal{L}_0}, x_{\mathcal{L}_1}, .. x_{\mathcal{L}_{\ell-1}})$ and $(x_{\mathcal{R}_0}, x_{\mathcal{R}_1}, .. x_{\mathcal{R}_{\ell-1}})$; compute the above information for j, $(x_{\mathcal{L}_0}, x_{\mathcal{L}_1}, .. x_{\mathcal{L}_{\ell-1}})$ and $(x_{\mathcal{R}_0}, x_{\mathcal{R}_1}, .. x_{\mathcal{R}_{\ell-1}})$ all of which takes constant time. From here we check if the inverted repeat can be extended to the first black position using the computed information. Each time a black position is reached it is checked if there is a match between the black position and the corresponding position of the inverted repeat. If there are multiple matches the one which maximises the probability is chosen. From here continue trying to extend the inverted repeat past black positions until one of the following becomes true:

- Neither $\mathcal{L}$ nor $\mathcal{R}$ are black positions.
- The $\log z / \log(\frac{z}{z-1})$+1-th black position of the inverted repeat is reached.
- $\mathcal{L} = 0$ or $\mathcal{R} = n - 1$.
- $x[\mathcal{L}] \cap \bar{x}[\mathcal{R}] = \emptyset$.

After this processing the longest inverted repeat centred at j with at most $\log z / \log(\frac{z}{z-1})$ black positions has been computed. This processing does not take into account the actual probability of the characters at black positions, nor does it account for any grey positions. This inverted repeat can be reported as it's quadruple (i, r, e, b), note that $b = \mathcal{O}(1)$.

The problem now is to determine how long the maximal valid inverted repeat actually is. To determine the maximal valid inverted repeat a data structure is constructed which allows us to determine the maximum radius of a valid inverted repeat at a given centre in constant time. From here it is a assumed that any preprocessing associated with extended factors is piggybacked on to the construction of the weighted suffix tree.

To determine the maximum radius of an inverted repeat, an array M is computed which for each index $0 \leq i < n$, $\mathsf{M}[i]$ stores a list with the maximum radius of valid factors with centre i. Let $\mathcal{E}$ be the set of all extended factors such that $|\mathcal{E}| = s$ with the extended factors arbitrarily labelled from 0 to $s-1$. An extended factor with label α is denoted by u_α. When referring to an extended factor it is indexed by the position it occurs in the original weighted string. For an extended factor u_α which is a factor of the weighted string occurring at position i, the extended factor is indexed as $u[i \,..\, i + |u_\alpha| - 1]$ rather than $u_\alpha[0 \,..\, |u_\ell| - 1]$ and for any extended factor u_α denote the start position by i_{u_α}. Let $\mathcal{B}_{u_\alpha}$ be the set of pairs (v, a) such that v is a black

position and $u_\alpha[v] = a$ for some factor u_α. We define an array M_α for $0 \leq \alpha < s$ for $u_\alpha \in \mathcal{E}$ which gives the maximum radius of an inverted repeat for each position in the extended factor u_α. More formally M_α and M are defined as follows for all $i_{u_\alpha} \leq i < i_{u_\alpha} + |u_\alpha|$ and $0 \leq \alpha < s$.

$$\mathsf{M}_\alpha[i] = \{\ (j,\ \mathcal{B}_{u_\alpha[i-j\,..\,i+j-1]})\ |\ \mathsf{max}\{j : \pi_{i-j}(u_\alpha[i-j\,..\,i+j-1]) \geq 1/z\}\}$$

M is then defined as the union over all M_α for each $0 \leq i < n$ as seen below:

$$\mathsf{M}[i] = \bigcup_{\alpha=0}^{s-1} \mathsf{M}_\alpha[i]$$

This is computed during the construction of the weighted suffix tree by generating the extended factors and then, for each position in an extended factor, find the length of the longest valid factor starting at that position; from this M_α can be computed as follows. For the longest valid factor v starting at some position, determine the midpoint t, of the factor and insert the value into the list $\mathsf{M}_\alpha[t] = \lfloor \frac{|v|}{2} \rfloor$. After computing all midpoints any missing values can be computed by taking the index of the closest previous and next already computed values o and p, and for all $(\beta, \gamma) \in \mathsf{M}_\ell[o]$ we add $(o + \beta - i, \gamma)$ to $\mathsf{M}_\alpha[i]$ and for $(\beta', \gamma') \in \mathsf{M}_\alpha[p]$ we add $(\beta' - p + i, \gamma')$ to $\mathsf{M}_\alpha[i]$.

Lemma 44. *M_α is correctly computed for $0 \leq \alpha < s$; therefore M is correctly computed.*

Proof. Correctness is clear for those entries which are directly added as midpoints, it suffices to show that the missing entries are correctly computed. Consider the longest valid factors starting from position i and $i+1$ of length u and u' such that the midpoints differ by at least two. The midpoint of consecutive positions can only increase so those in the difference between the midpoint of i and $i+1$ will neither have entries already nor be filled later when other positions are considered. For one of the undefined positions let o be the midpoint of the factor starting at i and p the midpoint of the factor starting at $i+1$, assume that the centre point is not defined by $(o+\beta-i, \gamma)$ for $(\beta, \gamma) \in \mathsf{M}_\ell[o]$ or $(\beta' - p + i, \gamma')$ for $(\beta', \gamma') \in \mathsf{M}_\alpha[p]$. If this is the case it means that it factor that defines the midpoint extends past $i+1$ and the first valid factor was not maximal which is a contradiction. Otherwise it is contained entirely within a valid factor. □

Given all this information we can determine the maximum valid radius for any centre, so can determine the maximal valid inverted repeat. Given

the maximal inverted repeat with $\log z / \log(\frac{z}{z-1})$ black positions (i, r, e, b) we iterate through $\mathsf{M}[i]$ and for each entry (j, q) we check if $q \subset b$. Once we find an entry satisfying this we shorten the inverted repeat if $r > j$. Each of the checks mentioned take constant time, however, it is unclear how large $\mathsf{M}[i]$ could be. Finally, to achieve the desired complexity we show that $|\mathsf{M}[i]| = \mathcal{O}(1)$.

Lemma 45. *The list $\mathsf{M}_\alpha[i]$ contains at most a constant number of entries.*

Proof. For a maximal valid factor inserted from position y until position t the midpoint is defined as $y + \frac{t-y}{2}$. We have two cases for any midpoint, either it is an integer, or it is not. In the case that it is an integer we directly add $\mathsf{M}_\alpha[y + \frac{t-y}{2}] = \lfloor \frac{t-y}{2} \rfloor$. In the case it is not an integer it could be the midpoint for two values and we update as follows $\mathsf{M}_\alpha[y + \lfloor \frac{t-y}{2} \rfloor] = \lfloor \frac{t-y}{2} \rfloor$ and $\mathsf{M}_\alpha[k + \lceil \frac{t-k}{2} \rceil] = \lfloor \frac{t-k}{2} \rfloor$. Now consider three consecutive valid factors from the same extended factor defined by their start and end points (y, t), $(y+1, t')$ and $(y+2, t'')$. Assume we have correctly processed up to position $y - 1$. We know that $t'' \geq t' \geq t$, therefore $\frac{t''-y+2}{2} > \frac{t'-y+1}{2} > \frac{t-y}{2}$ from this we have $\lfloor \frac{t''-y+2}{2} \rfloor > \lfloor \frac{t-y}{2} \rfloor$ and $\lceil \frac{t''-y+2}{2} \rceil > \lceil \frac{t-y}{2} \rceil$.

We see that position y and $y+2$ must contribute to a different midpoint list and that the mid point list that position $y+2$ is added to must be a higher index. Consider the size of the mid point list that $y+1$ contributes to. It may be the same as either the list y or $y+1$ contributes to, but no other index in this extended factor can add to the list; therefore, the size is constant. Clearly the entries which are not directly filled out cannot exceed the size of the two nearest already filled entries; so the size is constant. □

Combining Lemma 39 and 45 it can be seen that the size of $\mathsf{M}[i]$ is constant for all $0 \leq i < n$.

Theorem 46. *Problem 30 can be solved in runtime $\mathcal{O}(n)$.*

Weighted Inverted Repeats with k-mismatches

Computing valid inverted repeats with k-mismatches naively seems as though ever lce query may start from a black position and it this could lead to σ^{k+1} queries. Additionally, depending on how the probabilities are distributed at black positions there may be situations where the maximal inverted repeat requires us to take a mismatch at a black position, even though a match exists, if it keeps the cumulative probability significantly higher and leads to a longer inverted repeat.

The approach presented for the exact case generalises quite easily for the case of k-mismatches. All of the initial steps are identical, when we get to the point in the algorithm where the following would be computed:

- Compute $\mathsf{lce}(x[i+1], \bar{x}^R[n-i-1])$.
- Compute $\mathsf{lce}(x[2i - x_{L_r} + 2], \bar{x}^R[n - x_{L_r} - 2])$, for $0 \leq r < \ell$.
- Compute $\mathsf{lce}(x[x_{R_r} + 1], \bar{x}^R[n - x_{R_r} + 2])$, for $0 \leq r < \ell$.

The difference is that rather than only one lce query in each case, we perform up to k and record where any mismatches occur. In each case we wish to determine an extension up until the nearest black position with at most k-mismatches. Again we start from a centre and try and extend outwards using the precomputed information to determine the maximal radius r. In this case we extend outwards from the centre until we find k-mismatches or reach the $\ell + 1$th black position. The position of each mismatch found is stored in a linked list K. In this case when we extend outwards we do not check for black positions yet, all mismatches will occur between white and grey positions only. Candidates for the maximal valid inverted repeat are now formed by generating all possible combinations of characters from the $\log z / \log(\frac{z}{z-1})$ black positions within radius r. Each combination of black positions is considered as a candidate for the maximal inverted repeat. A black position contains no more than z valid characters so there will be no more than $z^{\log z / \log(\frac{z}{z-1})} = \mathcal{O}(1)$ possible combinations. For each candidate start from the centre and again go outwards to each black position and determine if the character at each black position causes a match or a mismatch. If there is a mismatch and this causes the total number of mismatches to be greater than k, remove the last element of K and determine the new radius of the inverted repeat. This process stops when either: every black position has been processed or the next black position to be considered is outside of the radius of the inverted repeat. This clearly takes constant time for each candidate, so constant time in total. Now we iterate through $\mathsf{M}[i]$ for each candidate inverted repeat to determine if it is valid and if not, how to shorten it so it is valid; this is also a constant time operation. Finally take the maximum radius inverted repeat and we are done, from this we get the following:

Lemma 47. *For a position i, there are at most $\mathcal{O}(k)$ lce queries.*

Therefore,

Theorem 48. *Problem 31 can be solved in runtime $\mathcal{O}(kn)$.*

5.4 Repetitions in Weighted Strings

A fundamental structural characteristic of a string of letters is its periodicity. Closely related to periodicity is the notion of *repetition*. Repetitions in strings are highly periodic factors, that is, two or more adjacent identical factors. For instance, the string `TATA` is a repetition in the string `CTATAGT`. Clearly a string may contain a quadratic number of repetitions. In 1981, it was shown by Crochemore that there could be $\mathcal{O}(n \log n)$ *maximal repetitions* in a string of length n and an $\mathcal{O}(n \log n)$-time, thus optimal, algorithm was presented [37]. In 1999, Kolpakov and Kucherov presented an $\mathcal{O}(n)$-time algorithm to compute a most compact representation of all maximal repetitions known as *runs* [76].

Tandem duplication, in the context of molecular biology, occurs as a result of mutational events in which an original segment of DNA is converted into a sequence of individual copies. It usually results from replication slippage or from certain recombination events, such as unequal crossing-over or unequal sister chromatide exchange [28]. In this context, the result of a tandem duplication event is termed a *tandem repeat*. It appears in both eukaryotic [94] and prokaryotic [128] genomes.

Through time, individual copies within a tandem repeat may change by additional, uncoordinated, mutations, and so only approximate tandem copies may be present. The major bottleneck in identifying biologically relevant tandem repeats in genomic sequences is a certain variation threshold that must be admitted between the copies of the repeated segment. In other words, biologists are interested in *approximate* tandem repeats and not necessarily in exact tandem repeats only. A plethora of algorithms and tools for the identification of tandem repeats measuring this approximation have already been released (for instance, see [23], [75], and [25]).

The simplest and perhaps most widely-used notion for measuring this approximation is the notion of Hamming distance [77]. Another way of measuring this approximation is using a probabilistic model of biological sequences. Single nucleotide polymorphisms, as well as errors introduced by wet-lab sequencing platforms during the process of DNA sequencing, can occur in some positions of a DNA sequence. In some cases, these uncertainties can be accurately modelled as a *don't care* letter. However, in other cases they can be more subtly expressed, and, at each position of the sequence, a probability of occurrence can be assigned to each letter of the nucleotide alphabet; this process gives rise to a *weighted sequence*. For instance, consider a IUPAC-encoded [1] DNA sequence, where the ambiguity letter `M` occurs

at some position of the sequence, representing either base A or base C. This gives rise to a weighted DNA sequence, where at the corresponding position of the sequence, we can assign to A and C an occurrence probability of 0.5.

Research into efficient algorithms for computing regularities in weighted sequences was first initiated by the work of Iliopoulos *et al.* in [66] for *motif extraction* from weighted sequences. Various types of regularities in weighted sequences have been studied since. The authors of [65] proposed an $\mathcal{O}(n^2)$-time algorithm for computing all factors having two or more occurrences in a weighted string x of length n and later the authors of [32] proposed an $\mathcal{O}(n \log d)$-time algorithm for computing all factors of length d having two or more occurrences in x. The authors of [133] proposed an $\mathcal{O}(n^2)$-time algorithm for computing all *loose repeats*, that is, all factors having two or more occurrences that may have overlapping segments such that the overlapping part consists of different letters.

The efficiency of the proposed algorithms relies on the assumption of a given constant, the *cumulative weight threshold*, defined as the minimal probability of occurrence of factors in the weighted sequence. Recently, the authors of [135] proposed an $\mathcal{O}(n^2)$-time algorithm for computing all tandem repeats in a weighted sequence of length n.

5.5 Tandem Repeats Algorithm

To achieve the main result of the chapter we first solve a related sub-problem on the computation of *valid repetitions*. Additionally we define the notion of an *extended repetition* in the process. An *extended repetition* is a repetition occurring in an extended factor of x. A *valid repetition* $v = u^e$, $e \geq 2$, in x is defined as a quadruple (i, p, b, e) such that $u = v[0\,..\,p-1] = v[p\,..\,2p-1] = \ldots = v[(e-1)p\,..\,ep-1]$, where v is a *valid* factor of length ep of x occurring at position i; u^{e+1} is not a valid factor of x; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. We define the following subproblem.

Problem 49 (Extended Tandem Repeats in Weighted Strings)**.** *Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all valid repetitions in x.*

Lemma 50. *Every valid repetition in x occurs in at least one extended factor.*

Proof. Immediate from Lemma 36. □

For each generated extended factor, we run Crochemore's partitioning algorithm for maximal repetitions; the result is all the maximal extended repetitions in x. After computing all the maximal extended repetitions we cannot simply report all of these as valid repetitions. All valid factors must occur in an extended factor but extended factors may contain factors which are not valid. This is a consequence of treating grey positions as white during the generation of extended factors [64]. Since not all maximal extended repetitions are valid repetitions, we must therefore break up these maximal extended repetitions into valid repetitions to solve Problem 49.

In order to break up the maximal extended repetitions, we must compute some additional information. To determine how long any valid repetition should be, we must know, for each position i in an extended factor, the length of the longest valid factor starting at position i. The computation is based on the observation that the longest factor with probability greater than or equal to $1/z$ for the position $i+1$ has length greater than or equal to that of position i. To compute this we maintain an additional cumulative weight threshold π'. We store the computed lengths in an array LF of integers.

We start with the first position in an extended factor and naively compute the longest factor within the threshold by multiplying together the probability of the letters we encounter and storing this in π'. If multiplying the probability of some position $j > 0$ causes $\pi' < 1/z$ we set $\mathsf{LF}[0] := j - 1$. To proceed, we remove by division the occurrence probability of the first letter from π'. If $\pi' < 1/z$ then $\mathsf{LF}[1] = j - 1$; otherwise, we continue as before multiplying the probability of $j+1, j+2$, and so on, until the threshold is once again violated.

Example 51. *Let* $x = [(\mathtt{a}, 0.6)(\mathtt{c}, 0.4)]\mathtt{bab}[(\mathtt{a}, 0.6)(\mathtt{d}, 0.4)]\mathtt{bab}$ *and* $1/z = 1/2$. *The colouring corresponding to* x *is* `GWWWGWWW`. *The only extended factor generated by this weighted string is* `abababab`. *Starting at position 0, we set* $\pi' := 0.6$, *that is, the probability that* `a` *occurs at position 0. We then consider the probability of factors* `ab`, `aba`, *and* `abab`, *the probability of whose is also* 0.6. *Now we consider the next letter at position* 5 *and set* $\pi' := 0.36 < 1/2$. *We set* $\mathsf{LF}[0] := 4$, *and remove the probability of the first letter from the cumulative probability by setting* $\pi' := 0.36/0.6 = 0.6$. *Now we continue as before and consider the following factors (which now start from position 1)* `baba`, `babab`, `bababa`, *and* `bababab`. *We reach the end of the string without breaking the threshold, and so we set* $\mathsf{LF}[1] := 7$, $\mathsf{LF}[2] := 6$, $\mathsf{LF}[3] := 5$, *and so on until* $\mathsf{LF}[n-1] := 1$.

The sum of lengths of the extended factors is linear in n by Lemma 38.

The next step is to determine the set b for each maximal extended repetition.

This can be done in constant time per maximal extended repetition. We compute an array NB of integers of size n, such that for each position i in x, NB$[i]$ stores the index of the leftmost black position $j > i$; this can be done in linear time in n. For each maximal extended repetition u^e, we check all black positions in the first occurrence of u. There can only be a constant number of black positions in u; finding the black positions using NB takes time proportional to their number. It is now a simple case of recording the position and the letter present in the extended factor; this takes constant time per maximal extended repetition, so time proportional to the number of maximal extended repetitions in total.

Given all the maximal extended repetitions, we can now begin to break them up into valid repetitions. To achieve this, we can check the length of the longest factor starting at position i of the extended factor, and then determine the longest possible repetition starting from i. We can continue checking the maximal extended repetition in this manner reporting the length as we go. Note that in the worst case, for each maximal extended repetition u^e, we may check the starting position of each occurrence of u. As we show later (Lemma 53), this can be done efficiently. We now establish the maximal number of extended repetitions in x. Note that the work done by the algorithm so far is no more than the maximal number of extended repetitions.

Lemma 52. *There could be $\mathcal{O}(n \log n)$ extended repetitions in x.*

Proof. Consider the string partitioned into q non-overlapping segments N_i, $1 \leq i \leq q$, each of which contains $\ell = \lceil \log z / \log(\frac{z}{z-1}) \rceil$ black positions. For some segment N_i, each black position may generate $\mathcal{O}((z+1)^\ell)$ extended factors. By the definition of extended factors (see [64], for details), each extended factor may contain no more than ℓ black positions; so none of the extended factors can extend past the next segment N_{i+1}. Each of these extended factors may contribute to the number of extended repetitions. There may be no more than $\mathcal{O}(\ell(z+1)^\ell)$ extended factors from any N_i, and each is of length no more than $|N_i| + |N_{i+1}|$, so each extended factor may contribute $\mathcal{O}((|N_i| + |N_{i+1}|) \log(|N_i| + |N_{i+1}|))$ extended repetitions. Each segment may contribute no more than $\mathcal{O}(\ell(z+1)^\ell(|N_i| + |N_{i+1}|) \log(|N_i| + |N_{i+1}|)) = \mathcal{O}((|N_i| + |N_{i+1}|) \log(|N_i| + |N_{i+1}|))$ extended repetitions. Summing the extended repetitions that each segment may contribute, we achieve our claim that the number of extended repetitions is $\mathcal{O}(n \log n)$. □

As previously mentioned, whilst breaking some maximal extended repetition u^e into valid repetitions, we may need to check up to e positions. The maximum number of checks required will be the sum of the exponents of all

maximal extended repetitions returned by the partitioning. Now we establish the maximal sum of the exponents of maximal extended repetitions in a weighted string.

Lemma 53. *The sum of exponents of maximal extended repetitions in x is $\mathcal{O}(n \log n)$.*

Proof. Any primitive repetition u^e can also be seen as a sequence of overlapping primitive squares (as shown in Example 57). We know that the maximal number of occurrences of primitive squares is $\mathcal{O}(n \log n)$ [40]; clearly the sum of the exponents of primitive squares is also $\mathcal{O}(n \log n)$. By the definition of maximal extended repetitions each square is only in one maximal extended repetition. Therefore the sum of exponents of maximal extended repetitions is less than or equal to the sum of exponents of primitive squares so is also $\mathcal{O}(n \log n)$. □

Note that an analogous version of Lemma 52 holds for valid repetitions. We are now in a position to state our first result.

Theorem 54. *Problem 49 can be solved in optimal time $\mathcal{O}(n \log n)$.*

Proof. The proof of this can follow, almost identically, that of Lemma 52 but instead of considering extended repetitions we consider the time contributed by each segment; this too is $\mathcal{O}((|N_i| + |N_{i+1}|) \log(|N_i| + |N_{i+1}|))$ per segment. □

At this point, we have solved the subproblem which forms the basis for our solution. Intuitively, the subproblem finds repetitions $v = u^e$, where factor v occurs with probability greater than or equal to $1/z$. The idea behind our solution to Problem 32 is based on the observation that a repetition of exponent $e \geq 3$ is composed of overlapping occurrences of smaller repetitions. We intend to compute smaller repetitions and, from this, derive larger ones. Part of the process of computing valid repetitions was to break up maximal extended repetitions below the threshold into smaller valid repetitions. To determine the repetitions specified in Problem 32, we reverse this process and *compose* longer repetitions from small valid repetitions.

In order to solve Problem 32, we start by solving Problem 49 for threshold $k = 1/z^2$. The number of valid repetitions reported for k can be shown to be $\mathcal{O}(n \log n)$ by the same argument as for Lemma 52; and the number of black positions in a valid factor is only a constant amount higher than for the original threshold by a similar argument to the proof of Lemma 35. We pick $k = 1/z^2$ as we wish to guarantee that we will at least find squares such

that each half may have probability greater than or equal to $1/z$. We may also find repetitions with a higher exponent and repetitions which have a probability less than $1/z$, but we will explain how to filter these out using the same techniques as for Problem 49.

We alter the solution to Problem 49 to simplify the solution to Problem 32. Instead of breaking up maximal extended repetitions into valid repetitions, we break them into all their valid overlapping squares. There are no more than $\mathcal{O}(n \log n)$ valid squares by [40]. This can be shown by an almost identical argument as Lemma 52. To split maximal extended repetitions into their valid overlapping squares, we process them one by one and create a new square for each overlapping square in the maximal extended repetition. We only need to perform this on maximal extended repetitions of exponent $e \geq 3$, and this will take time proportional to the sum of the exponents which, by Lemma 53, is $\mathcal{O}(n \log n)$.

To perform the filtering step, we must check if both halves of the square are above the threshold $1/z$. To check each half, we compute, for each position i in an extended factor, the length of the longest valid factor starting at position i. During the generation of extended factors for the threshold k, we at the same time determine the longest factor with probability greater than or equal to $1/z$ by computing an array LF' which stores the analogous information. Filtering the squares in time proportional to their number can be done by checking that the length stored in the array is greater than or equal to the period of the square.

After the filtering step, we have a set of quadruples (i, p, b, e) representing all primitive squares such that each half of a square has a probability of occurrence at least $1/z$. Now, for every position i in x, we declare an array A_i of linked lists, such that the linked list $\mathsf{A}_i[f_i(j)]$, $f_i : [1, \lfloor n/2 \rfloor] \to [0, \mathcal{O}(\log_\phi n)]$, stores all the squares which occur at position i with period $j \in [1, \lfloor n/2 \rfloor]$. We now wish to establish the size of A_i and the size of the linked lists stored at any $\mathsf{A}_i[f_i(j)]$ We are now ready to establish the size of A_i.

Lemma 55. *A_i is of size $\mathcal{O}(\log_\phi n)$, where $\phi = (1 + \sqrt{5})/2$, and the size of any linked list $\mathsf{A}_i[f_i(j)]$ is $\mathcal{O}(1)$.*

Proof. There are at most a constant number of valid factors starting from position i and it is well known that a string can contain no more than $\log_\phi n$ prefixes that are squares [40]. By Lemma 39, position i is only in $\mathcal{O}(1)$ extended factors. The suffixes starting from i in each extended factor contain no more than $\log_\phi n$ prefixes that are squares; this achieves the first part of our claim. For the second part, it is enough to note that each suffix of an

extended factor starting from i, which there is $\mathcal{O}(1)$ of, can only contain one square of a given period. □

We can now construct the repetitions specified in Problem 32. For each position i, we iterate through the linked lists of array A_i. We iterate through each linked list $\mathsf{A}_i[f_i(p)]$, where p is the considered period. We process each square element $(i, p, b, e) \in \mathsf{A}_i[f_i(p)]$ to extend the corresponding square as much as possible, by checking for an occurrence of the square at position $i + p$. For a linear string, it is simple to determine this. For each pair of overlapping squares, the second half of the first square is the first half of the second square; so it suffices to check whether there exists a square at position $i + p$ with the same period.

Example 56. *Consider* $y = \texttt{ababab}$ *that contains the following primitive squares:* $(0, 2, 2)$*,* $(1, 2, 2)$*, and* $(2, 2, 2)$*; we wish to find the repetition* $(0, 2, 3)$*. We start at position* 0 *of* y *with* $(0, 2, 2)$ *and check if there is a square of period* 2 *starting at position* 2*. A matching square exists so we extend the repetition and check position* 4*. There is no square at position* 4 *so we report the repetition* $(0, 2, 3)$*.*

For weighted strings the approach is very similar, with the addition of a few, constant-time, checks. We must check, for each pair of overlapping squares, if the black positions from the first square match with the black positions from the second square. There is a constant number of black positions so this takes constant time. Each time we find such overlapping squares, we extend our repetition and delete the square at position $i + p$ from the corresponding list. As soon as we find a position where we cannot extend the repetition we stop. We continue doing this until we have found all repetitions.

Example 57. *Let* $x = \texttt{aab}[(\texttt{a}, 0.5)(\texttt{b}, 0.5)][(\texttt{a}, 0.5)(\texttt{b}, 0.5)]\texttt{ab}$ *and* $1/z = 1/4$*. We would like to report repetition* $v = \texttt{ababab}$*, defined by* $(1, 2, \{\emptyset\}, 3)$*. For* $i = 1$*, we iterate through the linked lists of array* A_1*. For* $p = 2$*, we iterate through the linked list* $\mathsf{A}_1[f_1(2)]$*. We find the square* $\texttt{abab}$*, defined by* $(1, 2, \{\emptyset\}, 2)$*. We check for an occurrence of the same square at position* $i + p = 1 + 2 = 3$*, and find* $(3, 2, \{(0, a), (1, b)\}, 2)$ *in* $\mathsf{A}_3[f_3(2)]$*. We have to check if the black positions from the first square match with the black positions from the second square. They do, so we extend our square to repetition* $\texttt{ababab}$*, defined by* $(1, 2, \{\emptyset\}, 3)$*, and delete the square at position* $i + p = 3$ *from the list* $\mathsf{A}_3[f_3(2)]$*.*

Each time we iterate through a linked list, a square may be added to the repetition we are extending; this takes constant time per list by Lemma 55.

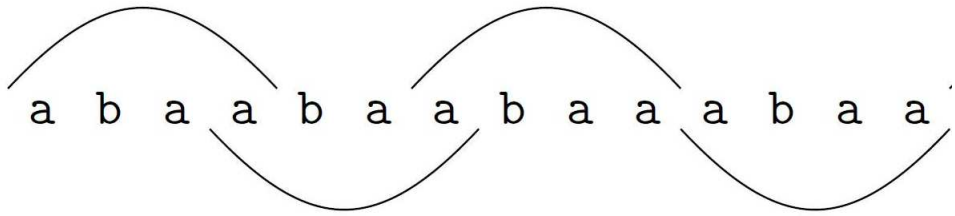


Figure 5.1: Cover of string *abaabaabaaabaa*

After each square is added to the repetition, it is deleted so is not considered again. There are $\mathcal{O}(n \log n)$ squares in the array and from the above description we can see that each square is considered a constant number of times. It is clear that we construct no more repetitions than there are primitive squares, so the number of constructed repetitions is also $\mathcal{O}(n \log n)$. These repetitions will be maximal, and to report repetitions specified in Problem 32, we may check the start of each occurrence in the repetition and report them. This takes no more than the sum of exponents which is $\mathcal{O}(n \log n)$. We can now state the main result of this section.

Theorem 58. *Problem 32 can be solved in optimal time* $\mathcal{O}(n \log n)$.

5.6 Covers in Weighted Strings

In this section we consider the computation of covers in weighted strings. Covers are a repetitive feature in strings distinct from both repetitions and periods. A factor w of a string x is called a cover of x if and only if x can be constructed by concatenations and superpositions of w; a cover of a string represents a generalised or relaxed notion of periodicity. Intuitively a factor of a string is only a cover if every position of the string is covered by at least one occurrence of the factor, this can be seen in Figure 5.1. Covers were first introduced by Apostolico *et al.* in [8], where they presented a linear-time algorithm to test the superprimitivity of a string and give its minimum cover. Since the introduction of covers their efficient computation and combinatorial properties have been extensively studied for regular strings. Breslauer [26] presented an online linear-time for the computation of not just the minimum cover of some string x but also the minimum cover of all its prefixes. Moore and Smyth [95] presented an algorithm for the linear-time computation of *all* the covers of a string, this was later [87] improved to an online linear-time algorithm for the computation of all covers of every prefix of a string.

A cover of a weighted string x can be defined as a (non-weighted) string w such that every position in x is covered by a valid occurrence of w. An example of a cover of a string can be seen in Example 5.1. The computation of covers in weighted strings has been been investigated by a number of different researchers. In [134] an $\mathcal{O}(n^2)$ algorithm was presented for the computation of all covers in a weighted string based on an $\mathcal{O}(n^2)$ algorithm for the computation of equivalence classes similar to that of Crochemore's repetitions algorithm. In [32] an $\mathcal{O}(n \log d)$ algorithm was presented to compute d length covers which was based on the Karp *et al.* algorithm for computing equivalence classes in a string.

In the rest of this section we present a second optimal algorithm to compute tandem repeats in weighted strings. This time we take a more well known approach to the computation of tandem repeats and present a generalisation of Crochemore partitioning that works efficiently in weighted strings. We improve the computation of the equivalence relation used in Crochemore's partitioning algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ and then show that this can be easily modified to also compute covers.

5.6.1 Partitioning Algorithm

We start by providing a brief description of Crochemore's partitioning algorithm in regular strings. For a factor w in y, the set of starting positions of all the occurrences of w in y gives us the *start set* of w. We define an equivalence relation $\approx_p$ at positions of y, such that $i \approx_p j$ if and only if $y[i \,..\, i+p-1] = y[j \,..\, j+p-1]$. Therefore, depending on the length of the factor, we get equivalence classes for each length p, for all $1 \leq p \leq n$. Equivalence classes for $p = 1$ are found by going through y once, and keeping the occurrences of each letter in separate sets. For larger p, we consider classes of the previous level to make a refinement on them, and calculate the classes of that level. So on level p, such that $1 < p \leq n$, we refine a class C with respect to a class D by splitting C in classes $\{i \in \mathsf{C} / i+1 \in \mathsf{D}\}, \{i \in \mathsf{C} / i+1 \notin \mathsf{D}\}$.

In order to achieve a good runtime, we do not use all classes for refinement; only classes of the previous level, which were split two levels before, are used. From those, we can omit the largest siblings of each family, and use only the small ones for the computation, this is known as the *smaller half trick*. We terminate the algorithm when all classes reach a singleton stage, that is, when they contain only one element. An example of this partitioning technique can be seen in Figure 5.2.

An issue with previous attempts at applying this scheme to weighted string is that they run directly on the weighted string and the initial equiv-

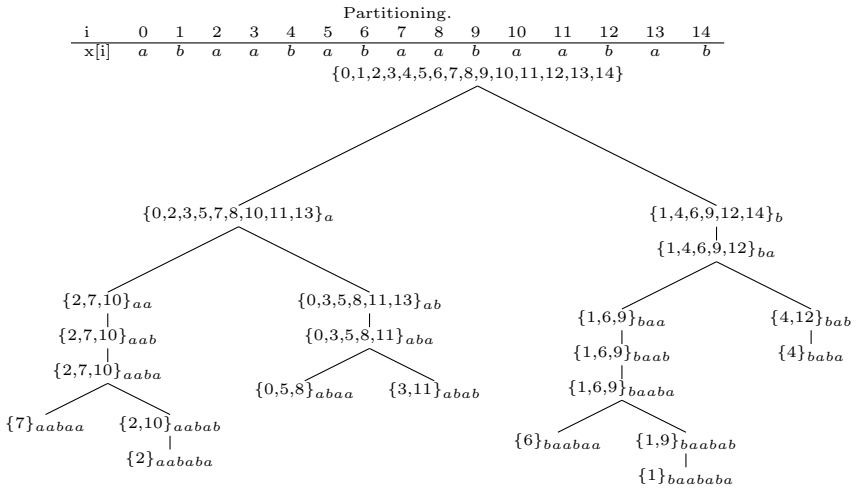


Figure 5.2: Classes of equivalence and their refinements for string $x = abaababaabaabab$.

alence classes only contain each position once. As a single position may generate many valid factors this means that they must duplicate positions at each step in the partitioning to ensure they do not miss valid factors. This is one issue that causes the problems mentioned in [33], that partitioning may generate $\mathcal{O}(\sigma^n)$ factors if the equivalence classes are not checked to prune out factors with probability below the threshold. Scanning the equivalence classes at each stage leads to a runtime of $\mathcal{O}(n^2)$. In the rest of this section we will describe how it it possible to efficiently apply this scheme to weighted strings.

The first step of the algorithm is to colour the weighted string and generate the extended factors as described in Section 5.1. Assuming this has been done, let $\mathcal{E}$ be the set of all extended factors such that $|\mathcal{E}| = s$ and assign a unique label to each from 0 to $s - 1$. From this point when referring to a position in an extended factor it is indexed by the position that it occurs in the original weighted string. For an extended factor u_ℓ starting at position i in the weighted string it is indexed as $u_\ell[i \,..\, i + |u_\ell| - 1]$ rather than $u_\ell[0 \,..\, |u_\ell| - 1]$. For an extended factor u_ℓ, where ℓ is its label, we denote it's start position in the weighted string by i_{u_ℓ}. Indexing like this can be achieved by associating to all extended factors their start position i_{u_ℓ}. This value can then be used as an offset for the index of the array.

Part of the efficiency of the partitioning scheme in regular strings is derived from the fact that once the initial classes are computed everything else can be done implicitly and there is no need for the original string in subsequent computation steps. We would like to maintain this property and to be able to partition implicitly. It has been shown in the previous sec-

tion that black positions prevent the start position and length from uniquely identifying a factor. Instead it is possible to uniquely identify factors by the triple: start position, length, and the label of the extended factor it is from. An extended factor implicitly represents the black positions contained within the factor and all combinations of black positions in the weighted string are represented by an extended factor. This way the need for the weighted string past the computation of the initial classes can be removed. In the following discussion we show that partitioning in this way means it can still be done implicitly.

The idea behind our approach is to generate the initial equivalence classes from the extended factors and then compute an equivalence relation on all extended factors simultaneously. The equivalence relation defined on regular strings is as follows:

$$(i,j) \in \mathsf{E}_p \text{ iff } (i,j) \in \mathsf{E}_{p-1}$$
$$\text{and } (i+1,j+1) \in \mathsf{E}_{p-1}$$

If the initial equivalence classes are computed over all extended factors according to the above relation there may be many copies of the same position in multiple equivalence classes and the equivalence relation no longer makes sense. This can be fixed by extracting the label of the extended factor as well as the position in the weighted string. This way the initial classes are generated over all extended factors and sets of tuples (i,h) are produced where i is the position in the weighted string and h is the label of the extended factor it was extracted from. The equivalence relation can now be redefined as follows:

$$((i,h),(j,k)) \in \mathsf{E}_p \text{ iff } ((i,h),(j,k)) \in \mathsf{E}_{p-1}$$
$$\text{and } ((i+1,h),(j+1,k)) \in \mathsf{E}_{p-1}$$

We further put the following restriction on each equivalence class:

$$\text{for all } (i,h) \in \mathsf{E}_p \ \pi_i(u_h[i\,..\,i+p-1]) \geq 1/z$$

This means that two tuples $((i,h),(j,k)) \in \mathsf{E}_p$ if and only if $u_h[i\,..\,i+p-1] = u_k[j\,..\,j+p-1]$, $\pi_i(u_h[i\,..\,i+p-1]) \geq 1/z$ and $\pi_j(u_k[j\,..\,j+p-1]) \geq 1/z$. Our solution is based on an efficient solution to the following problem:

Problem 59 (The Partitioning Problem)**.** *Compute* $\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_N$ *such that* $1 \leq N \leq n$ *and* $\mathsf{E}_N = \mathsf{E}_{N+1}$.

E_1 is computed by brute force from the extended factors by creating a tuple (j, k) for every position in every extended where j is the index in the weighted string and k is the label of the extended factor it was extracted from. For regular strings the equivalence relation is represented in two ways, an array E and a linked list ECLASS. The array E gives for each position the index of its current equivalence class. We modify the array to be two dimensional, for each extended factor u_ℓ there is an array with size $|u_\ell|$. Let $\{\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_q\}$ be the equivalence classes of E_p, we define E as follows for some tuple (j, k):

$$\mathsf{E}[j, k] = r \text{ iff } (j, k) \in \mathsf{C}_r$$

As before we index this array by the position the extended factors occur in the weighed string, this can be easily implemented by storing i_{u_k} for each extended factor and using this as an offset. The equivalence relations are also stored as a doubly linked list ECLASS which for each class $\mathsf{ECLASS}[i]$ stores a list of the positions in the class with index i. This is modified so rather than storing positions it stores tuples. We additionally store a two dimensional array of the same size as E but instead this new array EP stores for each tuple a pointer to the corresponding element in ECLASS. We compute E_1 in such a way that the elements of ECLASS are stored in increasing order of positions.

A difference function is defined on the tuples which will be used to efficiently compute the repetitions in an equivalence class in time proportional to their number:

$$\mathsf{D}_p = \mathsf{min} \begin{cases} \text{the least integer } g \geq 0 & \text{s.t } ((i, h), (i + g, k)) \in \mathsf{E}_p \text{ iff } h \neq k \\ \text{the least integer } g > 0 & \text{s.t } ((i, h), (i + g, k)) \in \mathsf{E}_p \text{ iff } h = k \\ \infty & \text{if no such integer exists.} \end{cases}$$

The difference function is represented in a similar way we the equivalence relation, a two dimensional array D and doubly linked list DCLASS such that $\mathsf{DCLASS}[p]$ gives the list of tuples which satisfy $\mathsf{D}[i, h] = p$ and DP which for each tuple points to it's corresponding element in DCLASS.

To compute the equivalence relation efficiently it is important to be able to quickly determine when a position should be excluded because its corresponding factor's probability is too low. For each position i in an extended factor, we compute the length of the longest valid factor starting at position i. This is exactly what is computed in the LF array in the previous section.

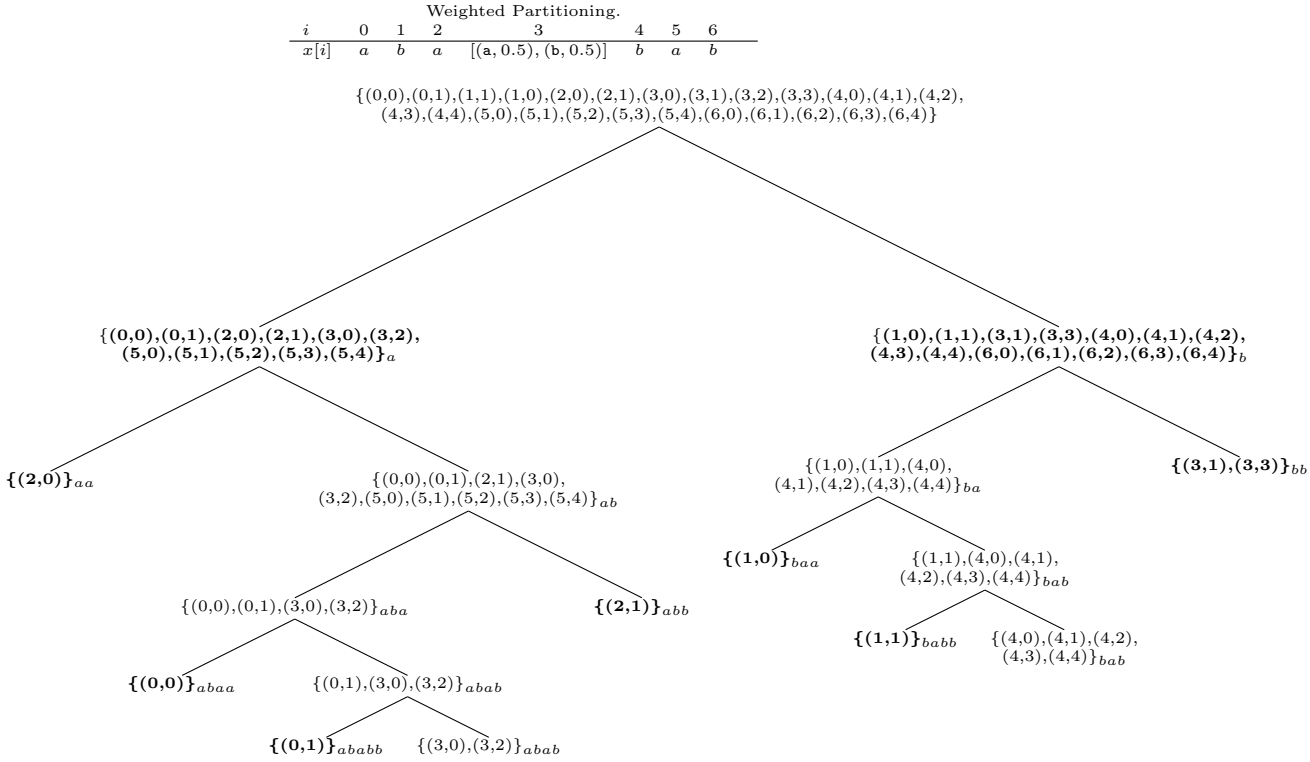


Figure 5.3: Classes of equivalence with their refinements for weighted string $x = aba[(a, 0.5), (b, 0.5)]bab$. The small sets of the partitioning are shown in bold.

An LF array is computed for each extended factor and denoted by LF_ℓ for the extended factor with label ℓ.

An additional array PR is now computed, let PR be an array of n linked lists and let $\mathsf{PR}[i]$ be the list of all pairs (j, k) such that $\mathsf{LF}_k[j] = i$, for $0 \leq k < s$ and $0 \leq i < n$. This array will specify the tuples which break the probability threshold this round and will be used to efficiently filter out such tuples after each partitioning step. It is important to filter out positions which break the probability threshold to ensure that the equivalence classes do not contain erroneous positions.

After each partitioning step, the appropriate index of the PR array is processed and the elements which violate the probability threshold are removed. The partitioning step corresponds to satisfying the first two conditions of the equivalence relation and processing the PR array satisfies the third, probabilistic, condition. Computing the equivalence relation directly can be realised in $\mathcal{O}(n^2)$, it remains to show that this can be done more efficiently than existing algorithms.

5.6.2 Correctness and Complexity Analysis

It was shown in [65] that if care is not taken in the partitioning and pruning steps then the algorithm will run in $\mathcal{O}(n^2)$. The algorithm of [65] starts with a small number of position in equivalence classes and duplicates them during partitioing, potentially leading to $\mathcal{O}(\Sigma^n)$ factors. This causes an important optimisation known as the *smaller half trick* to break down and no longer make sense. The smaller half trick provides the mechanism for efficiently refining an equivalence class. It tell us that when refining we do not need to inspect all classes, we may ignore the largest class resulting from some class being split.

The strategy to avoid the $\mathcal{O}(n^2)$ runtime is based on the generation of extended factors. By generating extended factors the size of the initial equivalence classes is increased but the need for duplication of positions is removed. This allows the smaller half trick and the original time complexity to be restored.

ALGORITHM WREP(x, n)
{Where x is a weighted string of length n.}
$\mathcal{E} \leftarrow$ extended factors;
compute PR;
define E to be E_1 on $\mathcal{E}$; define D to be D_1;
$p \leftarrow 1$;
while SMALL $\neq \emptyset$ **do**
  $p \leftarrow p + 1$; if $p > n/2$;
  $\mathsf{E} \leftarrow \mathsf{E} \cap \mathsf{S}'$;
  update D;
  SMALL $\leftarrow$ {indicies of small E classes};
  remove tuples stored in PR$[p]$;
end while

The first steps of the algorithm are to colour the string and to generate the extended factors. Clearly the colouring may be done in $\mathcal{O}(n)$ and by Lemma 38 we can see that the generation of the extended factors and therefore the size of the initial equivalence classes are both $\mathcal{O}(n)$. We know from Lemma 36 that every valid factor occurs within at least one extended factor, so computing the equivalence on all extended factors considers all valid factors. Lemma 38 also tells us that the computation of both the LF and PR array can be achieved in $\mathcal{O}(n)$ time.

Following the partitioning strategy defined in [37] we make a distinction

between *big* classes and *small* classes. Initially for $p = 1$ all classes are considered small, as a class is split during partitioning we consider all resulting class as *small* except the largest which is *big*. The idea behind the efficient computation of the equivalence relation is to only partition with respect to the small classes. This can be seen in the example below.

Example 60. *Consider the weighted string* $x = \mathtt{aba}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{bab}$ *with the cumulative probability threshold* $1/z = 0.25$*, it generates the following extended factors.* $\mathtt{abaabab}$ *and* $\mathtt{ababbab}$ *are generated from the first position;* $\mathtt{abab}$*,* $\mathtt{bbab}$ *and* $\mathtt{bab}$ *are generated from position 3. We assign them the following labelling* $(\mathtt{abaabab}, 0)$*,* $(\mathtt{ababbab}, 1)$*,* $(\mathtt{abab}, 2)$*,* $(\mathtt{bbab}, 3)$ *and* $(\mathtt{bab}, 4)$*. See Figure 5.3 to see the partitioning with the small classes in bold, we exclude classes from the tree when they are no longer considered for partitioning.*

The representation of the equivalence relation allows us to move an element from one class to another in constant time. We can directly access any tuple in the E array and change the index of it's equivalence class in constant time. The EP array can then be used to directly access any element in ECLASS and, as ECLASS is a linked list, move an element in constant time. We now show that partitioning with respect to the small classes is sufficient by showing that equivalent versions of Lemmas 3 and 4 from [37] hold for the case of tuples.

We define the following set of tuples:

$$((i,h),(j,k)) \in \mathsf{S}_p \text{ iff } \begin{cases} ((i,h),(j,k)) \in \mathsf{E}_p \\ \text{or both } (i,h) \text{ and } (j,k) \text{ are in big } \mathsf{E}_p \text{ classes} \end{cases}$$

Or equivalently:

$$((i,h),(j,k)) \in \mathsf{S}_p \text{ iff for any small } \mathsf{E}_p \text{ class } \mathsf{C}, (i,h) \in \mathsf{C} \text{ iff } (j,k) \in \mathsf{C}$$

It now suffices to show the following lemma over tuples of positions.

Lemma 61. *For any* $p \geq 1$*,* $((i,h),(j,k)) \in \mathsf{E}_{p+1}$ *iff* $((i{+}1,h),(j{+}1,k)) \in \mathsf{S}_p$*.*

Proof. E_p is a refinement of S_p therefore $\mathsf{E}_{p+1} \subset \mathsf{E}_p \cap \mathsf{S}_p$. Let $((i,h),(j,k))$ be two positions such that

$$((i,h),(j,k)) \in \mathsf{E}_p \text{ and } ((i+1,h),(j+1,k)) \in \mathsf{S}_p$$

If $(i+1,h)$ is in a small E_p class then $(j+1,k)$ is in the same E_p class; so $((i,h),(j,k)) \in \mathsf{E}_p$. If $(i+1,h)$ is in a big E_p class the $(j+1,k)$ is also in a big E_p class. From $((i,h),(j,k)) \in \mathsf{E}_p$ we can deduce that $((i+1,h),(j+1,k)) \in \mathsf{E}_{p-1}$ so they are in the same big E_p class. So again we have that $((i,h),(j,k)) \in \mathsf{E}_{p+1}$. □

Lemma 61 establishes that the smaller half trick can still be used. Further Lemma 61 tells us that partitioning may be performed by computing the following:

Let $((i,h),(j,k)) \in \mathsf{S}'_p$ iff $((i+1,h),(j+1,k)) \in \mathsf{S}_p$

$$\mathsf{E}_{p+1} = \mathsf{E}_p \cap \mathsf{S}'_p$$

As a class is split, keep track of how many elements are contained within each new equivalence class. When the partitioning step has been completed the class with the maximum number of elements is determined and labelled as big, the rest are labelled small and the indexes of small classes stored in SMALL. Partitioning like this takes time proportional to the sum of the size of all the small classes. In the following analysis positions which might be excluded due to the pruning step are ignored and will be considered later.

Lemma 62. *Let R be the sum of the sizes of all small class, $R \leq \mathcal{O}(n \log n)$.*

Proof. Consider a position i in a small E_p class F and let F' be its E_{p-1} class. By definition of the small classes:

$$|\mathsf{F}| \leq |\mathsf{F}'|/2$$

Thus no position can be in more than than $\mathcal{O}(\log n)$ small classes since by Lemma 36 the E_1 class of i has a cardinality less than $\mathcal{O}(n) - m + 1$ where m is the number of distinct characters in x and by Lemma 36 there are $\mathcal{O}(n)$ positions so this proves the claim. □

There are a few final considerations: how to efficiently perform the pruning step so that each E_p class only contains those factors which have a probability of occurrence $\geq 1/z$; how to ensure the pruning does not cause problems for the partitioning and to ensure that the difference function D is correctly updated.

The need for a pruning step is a consequence of all valid factors being in extended factors but not all factors of an extended factor being valid factors. To realise the pruning step we make use of the PR array and the EP array. After each partitioning stage, process the corresponding index of the PR array. An element contained in the PR array can be removed from its E_p class in constant time by setting its class to -1 in the E array and removing it from the ECLASS using the EP array to access the element in constant time. The initial E_1 classes only have $\mathcal{O}(n)$ elements so the pruning step cannot do any more work than this.

A potential issue with this approach is that it seems as though removing elements midway through the algorithm may cause problems with subsequent partitioning steps. Consider some tuple (i, h) has been removed, we must be able to guarantee that partitioning with respect to position $(i-1, h)$ is still computed correctly. This problem can be overcome as if (i, h) was removed at stage p then it must be the case that either $(i-1, h)$ has already been removed or that $(i-1, h)$ will become invalid when considering stage $p+1$. The only reason $(i-1, h)$ must be included in the partitioning at stage $p+1$ is to guarantee the correct partitioning of the tuple $(i-2, h)$ which may be valid at stage $p+1$ but wont be at $p+2$. The E_{p+1} class that $(i-1, h)$ ends up in after partitioning is therefore irrelevant as it will be removed immediately after.

Now it remains to show that D and DCLASS are correctly computed with no change to the time complexity. Consider that during a partitioning step each tuple that requires moving from E_p class to another is performed one at a time. Let (i, h) and (j, k) be two tuples such that (j, k) is a tuple that has been moved and (i, h) is the tuple that preceded (j, k) before it was moved, then the following must be true:

$$\mathsf{D}[i, h] = \mathsf{D}[i, h] + \mathsf{D}[j, k]$$

Furthermore, let (ℓ, m) be the tuple preceding (j, k) in its new equivalence class, then the following holds:

$$\mathsf{D}[\ell, m] = j - \ell$$

Let (p, q) be the tuple that immediately follows (j, k), we get the following:

$$\mathsf{D}[j, k] = j - q$$

Updating the DCLASS can be done by transferring a tuple to a new DCLASS at the same time it is moved from an ECLASS. We update the effected tuples

D array entries as specified above and update the DCLASS by moving any tuple (i, j) that has been updated into DCLASS[D$[i, j]$]. This can be achieved in constant time by using the DP array. We have presented a new optimal $\mathcal{O}(n \log n)$ algorithm for the computation of tandem repeats in weighted string, additionally we have shown that equivalence classes can be efficiently computed in weighted strings and we get the following.

Theorem 63. *The partitioning problem in weighted strings can be solved in* $\mathcal{O}(n \log n)$.

5.6.3 Covering a Weighted String

We are now ready to present the adaptations of the above algorithm to the computation of all covers in a weighted string. With the correct and efficient computation of the equivalence relation, computing all the covers in a string is relatively easy. For an equivalence class C_r we can maintain a variable MG_r which stores the maximum gap between any two consecutive tuples in C_r where tuples are ordered by their first elements. More formally let $\mathsf{C}_r = \{(x_1, e_1), (x_2, e_2), .., (x_a, e_a)\}$ be a set of tuples sorted by their first element, then MG_r is defined as follows:

$$\mathsf{MG}_r = \mathsf{max}\{x_{i+1} - x_i\} \text{ for } 0 < i < a$$

The important observation for the efficient computation of MG_r is that it is monotonically increasing.

Let $\{\mathsf{C}_1, \mathsf{C}_2, .., \mathsf{C}_q\}$ be the equivalence classes of E_p, to compute all covers we are required to maintain a variable MG_r for each equivalence class C_r which contain a tuple of the form $(0, k)$, for any $1 \leq r \leq q$ and any k, as any cover must also be a prefix of the weighted string. The initial values of MG_r for the equivalence classes of E_1 are computed by brute force when the initial equivalence relation E_1 is computed. We are only required to update the value of an MG_r variable if the equivalence class C_r is split and $(0, k) \in \mathsf{C}_r$ for some k. When updating some MG_r there are two cases: where $(0, k)$ is moved into an new C class and where it is not. If during the splitting of an equivalence class a tuple $(0, k)$ is moved then we can maintain MG_r at the same time as the D array is updated. For each tuple (i, h) that is moved, D$[i, h]$ is updated and we can check if D$[i, h] > \mathsf{MG}_r$, if so we update MG_r. Similarly if during the splitting of a class some tuple $(0, k)$ is not moved we update MG_r as the other tuples are moved and D is updated.

At each step p in the partitioning we check each class C_r which contains

a $(0, k)$ tuple, it is well known that a string is only a cover if the following hold:

- $\mathsf{MG}_r \leq p$;
- $(n - p + 1, h) \in \mathsf{C}_r$ for any h.

There are only $\mathcal{O}(1)$ possible tuples of the form $(0, k)$ by Lemma 39 and the checks above only take constant time so the algorithm takes the same time as the partitioning and we get the following:

Theorem 64. *The all-covers problem in weighted strings can be solved in* $\mathcal{O}(n \log n)$.

5.7 Conclusions and Future Work

In this chapter we first developed a number of efficient algorithms for finding inverted repeats in weighted strings. We have developed techniques to efficiently compute exact inverted repeats and approximate inverted repeats under Hamming distance in weighted strings. These algorithms are the first algorithms for this problem applied to the case of weighted strings and have the same worst-case complexity as those for the case of non-weighted strings. For future work we will attempt to extend our technique to the case of detecting inverted repeats under edit distance.

We have then presented an optimal algorithm for computing all tandem repeats in weighted sequences. The algorithm presented improves on the time complexity of the best-known algorithm for computing all tandem repeats in weighted sequences from $\mathcal{O}(n^2)$ to an optimal $\mathcal{O}(n \log n)$. Then we presented a second optimal algorithm for computing tandem repeats and improved the computation of covers in weighted strings from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. In doing so we also showed that Crochemore's partitioning scheme can be used efficiently in weighted sequences, something which was previously thought not possible.

For future work, we intend on devising an algorithm to compute a most compact representation of all maximal repetitions in weighted strings similar to the one for regular strings [76] and the computation of *seeds*.

6

Pattern Matching with Wildcards

Pattern matching with wildcards is a string matching problem where the alphabet consists of letters which only match themselves and a special letter ϕ which matches every character in the alphabet, the *wildcard*. Given a text of length n and a pattern of length $m < n$ the problem then consists of finding all factors of the text that match the pattern. There exists a number of variants of this problem where the number of wildcards is bounded, wildcards are restricted to either the pattern or text, or wildcards may match more than one symbol and where wildcards are optional. In this chapter we only focus on the case where wildcards match a single letter.

Pattern matching with wildcards has practical applications for a number of problems in bioinformatics, they are primarily used to model *single nucleotide polymorphisms* (SNP). Single nucleotide polymorphisms are relatively common (around 1% or more) substitutions that occur within a population. SNPs have diverse uses in the identifications of diseases as well as being used in genome wide association studies as markers for gene mapping. It is well known that single as well as multiple SNPs can cause disease, so their detection is an important task. In addition to their practical relevance, problems related to wildcards are a well studied area of theoretical interest.

An early and important result in pattern matching with wildcards was the *Fast Fourier Transform* (FFT) based algorithm of Fischer and Paterson [45] with runtime time of $\mathcal{O}(n \log m \log \sigma)$. This algorithm was the first to exploit the similarities between string matching and integer multiplication. A subsequent study by Pinter [107] outlined a number of difficulties in designing wildcard algorithms; he illustrates the problem of intransitivity which prevents traditional string matching algorithms such as KMP [74] being used or easily modified when wildcards are present. After Fischer and Patterson, much work focused on improving the algorithms by removing the dependency on the alphabet size, with randomized $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log m)$ solutions being proposed by [69] and [71] respectively. Later, deterministic $\mathcal{O}(n \log m)$

solutions were proposed, first by Cole and Hariharan [36] and then simplified by Clifford and Clifford [34]. All of these algorithms make use of the theoretically fast computation of the FFT, in fact, this is the only technique known for theoretically fast worst-case algorithms for pattern matching with wildcards.

The indexing version of the problem has also been studied by numerous researchers. In [67] Iliopoulos and Rahman presented an index supporting queries in $\mathcal{O}(m+\alpha)$ where wildcards only occur in the pattern and $\mathcal{O}(m+\alpha \log\log n)$ in the case of optional wildcards in the pattern. Understanding the time complexity requires us to recall their notation, denote the pattern as $p = p_0\phi^{g_0}p_1\phi^{g_1}\ldots\phi^{g_{h-1}}p_h$ where ϕ^{g_i} represents a group of consecutive wildcards and $p_i \in \Sigma^+$. α is then defined as the sum of the number of matches of each p_i in t for $0 \leq i < h$. A short coming of this approach is that in the worst-case α may be $\Theta(hn)$. In [35] Cole *et al.* presented an index which given a text with k wildcards and an integer d, allows searching for any pattern with at most d wildcards. For a pattern containing $g \leq d$ wildcards, the matching takes $\mathcal{O}(m + 2^g \log^k n \log\log n + occ)$ time[1]; when wildcards are restricted to either the pattern or the text the query time becomes $\mathcal{O}(m+2^g \log\log n+occ)$ and $\mathcal{O}(m + \log^k n \log\log n + occ)$ respectively. A drawback of the index of Cole *et al.* is that once the index has been built it can only be used to search for patterns with at most d wildcards. Very recently a number of indexes were presented by Bille *et al.* [24] where they give a linear space index with query time $\mathcal{O}(m+\sigma^g \log\log n+occ)$ and a linear query time index with space complexity $\mathcal{O}(\sigma^{k^2} n \log^k \log n)$. These are then modified for variable length wildcards by reducing the problem to searching with optional wildcards. In the area of linear size indexes for strings with wildcards, Lam *et al.* [78] have presented indexes for a number of problems shown in Table 6.1, where p_i is the same as defined above, t_i is analogously defined for a text t of length n, $occ(u, v)$ denotes the number of occurrences of u in v, $\gamma = \Sigma_{j=1}^{\ell+1} occ(t_j, p)$, h is the number groups of consecutive wildcards in the text, g is the total number of wildcards in the text, $\beta = \min_{1\leq i\leq h+1}\{occ(p_i, t)\}$.

Succinct indexes have been presented in [125] with a space usage of $((2 + o(1))n \log\sigma + \mathcal{O}(n) + \mathcal{O}(d \log n) + \mathcal{O}(k \log k))$ bits for a text containing d groups of k wildcards in total; this index is based on an augmented compressed suffix array. The authors of [60] proposed a compressed index where wildcards can only occur in the text with space usage $nH_h + o(n \log\sigma) + \mathcal{O}(d \log n)$ bits, where H_h is the h-th-order empirical entropy ($h = o(\log_\sigma n)$) of the text.

[1]We use the notation of [78] as it is more understandable than [35].

Problem	Query Time
Wildcards in t	$\mathcal{O}(m \log n + \gamma + occ)$
Wildcards in p	$\mathcal{O}(m + h\beta)$
Wildcards in t and p	$\mathcal{O}(m \log n + h\beta + \gamma + occ)$
Opt wildcards in t	$\mathcal{O}(m^2 \log n + m \log^2 n + \gamma \log n + occ)$
Opt wildcards in p	$\mathcal{O}(m + gh\beta)$
Opt wildcards in t and p	$\mathcal{O}(m^2 \log n + m \log^2 n + gh\beta + \gamma \log n + occ)$

Table 6.1: Properties of the indexes presented in [78]

The rest of this chapter is structured as follows: Section 6.2 we present algorithms for the case where wildcards are only allowed in the text, then for the case where wildcards are only allowed in the pattern, in Section 6.3 we give a general lower bound in Section 6.4 we give concluding remarks and discuss future work. Let $\approx^{\phi}$ be equality with wildcards considered, note that this relation is not transitive like $=$ is.

In this chapter the problems we consider are the following:

Problem 65 (Wildcards in the Text)**.** *Given a text t of length n drawn from $\Sigma \cup \{\phi\}$, and a pattern p of length m drawn from Σ. Find all i such that $t[i \,..\, i+m-1] \approx^{\phi} p[0 \,..\, m-1]$.*

Problem 66 (Wildcards in Both)**.** *Given a text t of length n drawn from $\Sigma \cup \{\phi\}$, and a pattern p of length m drawn from $\Sigma \cup \{\phi\}$. Find all i such that $t[i \,..\, i+m-1] \approx^{\phi} p[0 \,..\, m-1]$.*

6.1 Background on Average-case Analysis

In this section we give some background information on the literature concerning average-case complexity. The term *average-case* has been used to refer to various different assumptions when discussing online pattern matching in strings, here we discuss these different assumptions and justify the model we use.

The primary difference we consider here are assumptions relating to the pattern. Some papers assume that the pattern is randomly drawn from the alphabet, whilst others consider that the pattern is arbitrary. An arbitrary pattern is assumed in [38, 132, 74], however, in later work such as [49, 51, 9, 50, 10, 100] the assumption of a random pattern is made. Perhaps the first notion of average-case optimality for string algorithms is in [74] where Knuth conjectured his algorithm was average-case optimal in

the following sense:

'*Patterns of length m exist for arbitrarily large m, such that an average of at least $cn(\log m)/m$ bits must be inspected for all large n.*'

Yao [132] considered this conjecture and proved a stronger statement:

'*We have demonstrated that, for almost any pattern of length m, and a random text of length n we must, on average examine $\log_\sigma m$ characters.*'

Note that Yao's result and Knuth's conjecture do *not* require that the pattern be random. Yao shows the existence of a set of patterns where string matching takes $\mathcal{O}(\frac{n \log_\sigma m}{m})$ on average in a uniformly random text. It has since become customary to analyse algorithms as if both the pattern and text are random however, as we can see, this was not always the case. For example in [50] an algorithm for multiple pattern matching is presented and analysed as if all the patterns are random. The same algorithm was later applied to a problem where the patterns are not random and it was shown that the original analysis of the algorithm is still valid, although under stricter conditions on the variables involved.

The difference between the two notions is that for arbitrary patterns the stated complexity must hold for *any* given pattern and a random text. However, for a random pattern the average is over all patterns and all texts, so there may exist patterns that perform significantly worse than the complexity suggests they should. We explicitly draw a distinction between these two notions and consider *average-case* complexity to be the case that the pattern is not random and *expected-case* complexity when the pattern is also random. Clearly the notion of average-case complexity as defined here is a stronger one than that of expected-case complexity.

A second consideration considered here that is more relevant to average-case complexity than worst-case complexity is the fixed or adaptive nature of the order the characters of the text are inspected in. We refer to the order the characters are inspected in as the *inspection scheme*. In [132] Yao investigates the effect that the inspection scheme has on the average-case runtime of algorithms for exact matching. In particular, Yao explores how having a predetermined inspection scheme negatively effects the runtime when compared with an adaptive, pattern dependent inspection scheme. The algorithm we consider in this chapter has the property that, for each window of the text considered, characters are inspected in a fixed order dependent

only on m; this is a ubiquitous property in string algorithms. In [132] it is shown that, for classical string matching, inspecting the text in a fixed order prevents any such algorithm being average-case optimal when m and n are sufficiently close. In this chapter we explore the effect that this property can have on the average-case performance of algorithms for pattern matching with wildcards and refer to algorithms which examine the character inside a window in a fixed order as *fixed algorithms*. For the purpose of showing lower bounds in this chapter we consider a simplified model of computation for fixed algorithms and define them as follows.

Definition 67 (Fixed Algorithm)**.** *Consider t partitioned into non-overlapping sections of size $2m$. Let $(i_1, i_2, .., i_{2m})$ be an arbitrary but fixed permutation of $(0, 1, 2, 3, .., 2m-1)$. An algorithm is fixed with respect to $(i_1, i_2, .., i_{2m})$ if, for every section, characters are inspected in the order specified by $(i_1, i_2, .., i_{2m})$. The problem is then to find all factors of t that correspond to p and are contained entirely within a section.*

We only consider occurrences contained entirely within a section to simplify the analysis. Clearly this lower bounds the procedure to find all occurrences by examining the characters of a sliding window in a fixed order. We show a tight upper bound on the best performance for these types of algorithms and for *non-fixed* algorithms we show a lower bound which we only know is tight for large g. Additionally we show that a greedy inspection scheme is optimal. We consider the following simplified definition of *non-fixed* algorithms.

Definition 68 (Non-fixed Algorithm)**.** *Consider t partitioned into non-overlapping sections of size $2m$. An algorithm is non-fixed if characters of t can be inspected in any order. The problem is then to find all factors of t that correspond to p and are contained entirely within a section.*

It is important to point out that when discussing the average-case complexity of these online string matching problems it is customary to make a distinction between time taking to preprocess the pattern and the *search time*. Additionally in the literature *average-case optimal* customarily refers to achieving the optimal search time, not necessarily considering the preprocessing time required to achieve it.

Our Contribution: In this chapter, we present fast average-case algorithms for pattern matching with wildcards and explore a number of models and assumptions surrounding average-case complexity. We establish lower bounds for the considered problems and present algorithms with average-case search

time $\mathcal{O}(\frac{n \log_\sigma m}{m})$ and $\mathcal{O}(\frac{n(g+\log_\sigma m)}{m})$ for Problems 65 and 66 respectively. Additionally we show both of these algorithms have expected-case search time $\mathcal{O}(\frac{n \log_\sigma m}{m})$, show a lower bound of $\mathcal{O}(\frac{n \log_\sigma m}{m-g})$ for the average-case search time of Problem 66 in the non-fixed model and show an optimal strategy for inspecting characters in the non-fixed model.

6.2 Algorithms

In the following section we present a number of average-case algorithms for pattern matching in the presence of wildcards. All of the algorithms follow the same scheme but have different details; the scheme is outlined below.

- Build a dictionary of the pattern for all the factors of length r over either $\Sigma \cup \{\phi\}$ or Σ, depending on the problem being considered.
- Create a sliding window of size m, then for each window we do the following.
- Check the suffix of size r, if it matches any factor of the pattern perform a naive $\mathcal{O}(m^2)$ algorithm verification algorithm on the window of size $2m$.
- Shift the window by $m - r$ positions if the suffix of size r does not match and m if it does.

The verification algorithm mentioned above consists of naively checking all possible alignments of the pattern against the text. Clearly each check takes no more than $\mathcal{O}(m)$ time and there are m possible start positions for a window of size $2m$ so $\mathcal{O}(m^2)$ in total. For the rest of the chapter assume that the text t is of length n and is random and uniformly drawn $\Sigma \cup \{\phi\}$ and that Σ is a finite but not necessarily constant alphabet. For each problem we design an algorithm according to the above scheme and analyse it in a number of settings. We determine the average-case complexity of the algorithm for an arbitrary pattern and then the expected-case complexity under the assumption that the pattern is also random.

6.2.1 Wildcards in Text Only

We begin by considering the problem where wildcards may appear only in the text and refer to the algorithm of this section as Algorithm Wt. The

scheme of the algorithm will follow the algorithm described at the start of this section. First we will establish the size of r, the suffix we will check for each window of the text and the probability of a match.

Lemma 69. *The probability of a random string of size* $3\log_{\frac{\sigma+1}{2}} m$ *over* $\Sigma \cup \{\phi\}$ *matching in a string of size* m *over* Σ *is at most* $\frac{1}{m^2}$.

Proof. Clearly for an alphabet of size σ the probability that a character of Σ matches a randomly picked character of Σ is $\frac{1}{\sigma}$. However, the probability of a character from Σ matching a randomly picked character of $\Sigma \cup \{\phi\}$ becomes $\frac{2}{\sigma+1}$. The probability of r characters matching in this way is given by $(\frac{2}{\sigma+1})^r$, so by setting $r = 3\log_{\frac{\sigma+1}{2}} m$ we get that $(\frac{2}{\sigma+1})^r = \frac{1}{m^3}$. There are at most $m - 3\log_{\frac{\sigma+1}{2}} m + 1$ factors of this length in a string of length m and so the probability of occurrence is certainly at most $\frac{1}{m^2}$. □

We now describe how the dictionary of factors will be built. We generate all strings of length r from the alphabet $\Sigma \cup \{\phi\}$, and check each against the set of r length factors from the pattern. Since there are $(\sigma+1)^r$ different generated strings, at most m factors of length r in the pattern and each check requires $\mathcal{O}(r)$ time, the total time spent is $\mathcal{O}(mr(\sigma+1)^r)$. We can store the results of this processing in a binary array where each string is converted to a numerical representation for efficient lookup. A dictionary built in this way has a search time of $\mathcal{O}(r)$. For this algorithm we set $r = 3\log_{\frac{\sigma+1}{2}} m$ and therefore guarantee $\mathcal{O}(\log_\sigma m)$ search time in the worst-case.

We can see that for $r = 3\log_{\frac{\sigma+1}{2}} m$ this is a polynomial preprocessing scheme and the maximum exponent occurs when $\sigma = 2$. For $\sigma = 2$ we have that $3^{3\log_{1.5} m} \approx m^{8.13}$ and so the total preprocessing is $\mathcal{O}(m^{9.13}\log m)$. For larger alphabets this exponent is greatly reduced.

We now present our algorithm for pattern matching with wildcards only in the text. Consider a sliding window of length m placed over the text and that for each window we will check if the suffix of length $3\log_{\frac{\sigma+1}{2}} m$ corresponds with any factor of the pattern. If the suffix corresponds with a factor of the pattern there may be a match and we are required to verify this window. To verify the window we run the $\mathcal{O}(m^2)$ verification algorithm. For each window on the text we have a total possible running time of $\mathcal{O}(m^2 + \log_{\frac{\sigma+1}{2}} m)$. The probability of the suffix corresponding with a factor of the pattern is $1/m^2$ by Lemma 65, so the expected time spent at each window is $\mathcal{O}(\log_{\frac{\sigma+1}{2}} m)$ or $\mathcal{O}(\log_\sigma m)$. If the window is verified then it is possible to shift by m characters, otherwise the window is not verified and can we shift by $m - r$ characters. This means there there are at most $\frac{n}{m-r}$ windows, each which has

an expected runtime of $\mathcal{O}(\log_\sigma m)$, giving us an overall average-case running time of $\mathcal{O}(n \log_\sigma m/m)$. From the above discussion we get the following result:

Theorem 70. *Algorithm* Wt *has average-case search time* $\mathcal{O}(n \log_\sigma m/m)$ *with* $\mathcal{O}((\sigma+1)^{3\log_{\frac{\sigma+1}{2}} m} m \log_\sigma m)$ *preprocessing time and* $\mathcal{O}((\sigma+1)^{3\log_{\frac{\sigma+1}{2}} m})$ *space.*

We now show that the search time achieved by Algorithm Wt achieves optimal average-case search time.

Theorem 71. *Algorithm* Wt *has average-case search time* $\mathcal{O}(n \log_\sigma m/m)$ *and achieves optimal average-case search time.*

Proof. Recall the lower bound for exact string matching is $\Omega(n \log_\sigma m/m)$ [132]. Clearly pattern matching with wildcards in the text only is at least as hard as exact string matching as we must also find non-wildcard matches. Algorithm Wt runs in average-case time $\mathcal{O}(n \log_\sigma m/m)$, matching the lower bound for exact string matching. Therefore, the algorithm is optimal and the lower bound on pattern matching with wildcards in the text only is also $\Omega(n \log_\sigma m/m)$. □

In this section we gave an upper bound and an average-case optimal algorithm for pattern matching with wildcards only in the text. In this situation we only require polynomial preprocessing and the lower bound on searching matches that of exact string matching within a constant factor. In this case the average-case and expected-case complexities match.

6.2.2 Wildcards in both the Pattern and the Text

In this section we consider the case where wildcards may appear in both the pattern and text. We first consider the average-case complexity. In this situation we can pick any pattern and for a pattern p we denote the number of wildcards in the pattern by g. We refer to the algorithm of this section as Algorithm Wp. We again follow the approach outlined at the beginning of this section; we now establish the size of the suffix we will check in the lemma below.

Lemma 72. *The probability of a random string of size* $g + 3\log_{\frac{\sigma+1}{2}} m$ *over* $\Sigma \cup \{\phi\}$ *matching in a string of size* m *over* $\Sigma \cup \{\phi\}$ *with* g *wildcards is no more than* $\frac{1}{m^2}$.

Proof. Consider a random factor of size $g + 3\log_{\frac{\sigma+1}{2}} m$ drawn from $\Sigma \cup \{\phi\}$. In the worst case all g wildcards of the pattern appear in a single factor of size $g + 3\log_{\frac{\sigma+1}{2}} m$, this factor contains $3\log_\sigma m$ positions which are not wildcards and the probability that these match the corresponding characters in a random factor of size $g + 3\log_{\frac{\sigma+1}{2}} m$ of $\Sigma \cup \{\phi\}$ is no more than $\frac{1}{m^3}$. For any factor with less than g wildcards the probability is less than $\frac{1}{m^3}$. Pessimistically assume all factors of length $g + 3\log_{\frac{\sigma+1}{2}} m$ have a probability to match of $\frac{1}{m^3}$, there are $m - (g + 3\log_{\frac{\sigma+1}{2}} m)$ factors so the probability is certainly no more than $\frac{1}{m^2}$. □

The dictionary we use for this algorithm is built in the same way as that of the previous algorithm, but for this problem we have that $r = g+3\log_{\frac{\sigma+1}{2}} m$. This means that the space complexity of the dictionary becomes $\mathcal{O}((\sigma + 1)^{g+3\log_{\frac{\sigma+1}{2}} m})$ with preprocessing time $\mathcal{O}((\sigma+1)^{g+3\log_{\frac{\sigma+1}{2}} m} m(g+\log_{\frac{\sigma+1}{2}} m))$.

Before we continue with the description of the algorithm we recall some notation from the literature on average-case approximate string matching and adapt if for our purposes. When considering string matching with k differences the tolerance of an algorithm to values of k is often expressed as an *error ratio* k/m, where k is the number of errors allowed and m the length of the pattern. Analogously, by g/m we denote the *wildcard ratio* of our algorithm. We now describe our algorithm for pattern matching with wildcards in both the pattern and text.

For this algorithm we create a sliding window on the text of length m. For each window on the text we check the suffix of length $g+3\log_{\frac{\sigma+1}{2}} m$ and if it corresponds to a factor of the pattern an $\mathcal{O}(m^2)$ verification algorithm is run on a factor of length $2m$. If the suffix does not correspond to a factor of the pattern we shift by $m - r$ and m if it did correspond we shift by m. As the minimum shift the algorithm makes is $g + 3\log_{\frac{\sigma+1}{2}} m$ there will be at most $\frac{n}{m-g-3\log_{\frac{\sigma+1}{2}} m}$ windows on the text and at each window we may do $\mathcal{O}(m^2 + g + \log_{\frac{\sigma+1}{2}} m)$ work in the worst case. The probability that we will need to verify a window is $1/m^2$ by Lemma 72, this gives us an expected time of $\mathcal{O}(g + \log_{\frac{\sigma+1}{2}} m)$ per window. Combining the expected work at each window with the number of windows we get $\mathcal{O}(n(g + \log_{\frac{\sigma+1}{2}} m)/m)$ in total.

For the algorithm to achieve the claimed runtime it must be the case that $\frac{n}{m-g-3\log_{\frac{\sigma+1}{2}} m} = \mathcal{O}(\frac{n}{m})$. To satisfy this it follows that $g + 3\log_{\frac{\sigma+1}{2}} m < \epsilon m$ where $\epsilon < 1$ and this places the following condition on the wildcard ratio of our algorithm:

$$\frac{g}{m} < \epsilon - \frac{3\log_{\frac{\sigma+1}{2}} m}{m}$$

We also have an additional restriction, we must be able to guarantee that we are reading enough new random characters after each shift that Lemma 72 still holds. This places the additional restriction that the window must be at least twice the length of the shortest shift. So it must hold that $m > 2g + 6\log_{\frac{\sigma+1}{2}} m$ to ensure that in all cases we only ever ready new characters as the suffix is read. After rearrangement this places the following restriction on our algorithm:

$$\frac{g}{m} < \frac{1}{2} - \frac{3\log_{\frac{\sigma+1}{2}} m}{m}$$

Clearly the second condition places the strictest condition on the wildcard ratio than the first. From the above discussion we achieve the following result:

Theorem 73. *Algorithm Wp has average-case search time* $\mathcal{O}(n(g+\log_{\frac{\sigma+1}{2}} m)/m)$ *with* $\mathcal{O}((\sigma+1)^{g+3\log_{\frac{\sigma+1}{2}} m} m(g+\log_{\frac{\sigma+1}{2}} m))$ *preprocessing time and* $\mathcal{O}((\sigma+1)^{g+3\log_{\frac{\sigma+1}{2}} m})$ *space, for* $\frac{g}{m} < \frac{1}{2} - \frac{3\log_{\frac{\sigma+1}{2}} m}{m}$.

The wildcard ratio we specify is quite permissive. To see this note that for any ratio $g/m < 1/2$ it is possible to pick a sufficiently large value of m such that the algorithm can run in the claimed running time. In the following theorem we show that for any fixed algorithm it is impossible to do any better than this. We now show that for any integer g, there exists a lower bound of $\Omega(\frac{ng}{m})$ character inspections for fixed algorithms.

Theorem 74. *Algorithm Wp has average-case search time* $\mathcal{O}(n(g+\log_\sigma m)/m)$ *and no fixed algorithm can do better.*

Proof. Recall that the lower bound for exact string matching is $\Omega(n\log_\sigma m/m)$. We now show that any fixed algorithm has a lower bound of $\Omega(\frac{ng}{m})$ in the case where there are more than $\mathcal{O}(\log_\sigma m)$ wildcards in the pattern. Clearly when $g \leq \mathcal{O}(\log_\sigma m)$ the bound of Yao [132] holds as this problem requires us to report non-wildcard matches as well. If there is no occurrence of the pattern then we must check at least g characters before we can declare there is no match.

Assume the text is partitioned into non-overlapping blocks of size $2m$ and that we only want to find occurrences contained entirely within these blocks.

Let π^{2m} denote all the permutations of $(0, 1, .., 2m-1)$ and assume that we examine the characters of each block in the same fixed but arbitrary order $(i_0, i_1, .., i_{2m-1}) \in \pi^{2m}$. We can construct patterns, for any $g < m$, such that we must examine at least $g+1$ characters before all start positions can be ruled out.

We construct a pattern in the following way, if for $0 \leq j < g$ all occur within a range of m character then we place the wildcards in positions $i_0, i_1, .., i_{g-1}$ of the pattern. Otherwise for $0 \leq j < g$ and $i_j < m$ place a wildcard at position i_j. Any remaining wildcards may be placed anywhere in the pattern; the remaining positions of the pattern are random characters from Σ. After inspecting characters $i_0, i_1, .., i_{g-1}$ of the block at least the first position can neither be ruled out nor declared as a match. Combining the lower bound of Yao and this we see that any fixed algorithm has a lower bound of $\Omega(n(g + \log_\sigma m)/m)$ for this problem. Algorithm Wp runs in average-case time $\mathcal{O}(n(g + \log_\sigma m)/m)$, matching the lower bound; therefore the algorithm is optimal in the family of fixed algorithms. □

Now we consider the expected-case complexity of the algorithms when the pattern is randomly drawn from $\Sigma \cup \{\phi\}$. By setting $r = 3\log_{\frac{(\sigma+1)^2}{3\sigma+1}} m$ the probability of a match remains $\frac{1}{m^2}$, the proof of this is essentially identical to that of Lemma 65. Similarly the proof of the runtime is same as the proof of Algorithm Wt with $r = 3\log_{\frac{(\sigma+1)^2}{3\sigma+1}} m$. Following this argument we get the following:

Theorem 75. *Algorithm Wp runs in expected-case search time* $\mathcal{O}(n \log_\sigma m/m)$ *with expected-case preprocessing time* $\mathcal{O}((\sigma+1)^{3\log_{\frac{(\sigma+1)^2}{3\sigma+1}} m} m \log_\sigma m)$ *and expected-case space usage of* $\mathcal{O}((\sigma+1)^{3\log_{\frac{(\sigma+1)^2}{3\sigma+1}} m})$.

Clearly it is the case that the expected-case complexity is expected-case optimal as it matches the lower bound of exact string matching. The difference between this and the average-case complexity suggests to us that although there exists difficult patterns, such patterns are rare.

6.3 A General Lower Bound

In the previous section we have considered the average-case and expected-case complexity of each algorithm. Where wildcards are allowed in both

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	ϕ	ϕ	b	ϕ	ϕ	a								
	a	b	ϕ	ϕ	b	ϕ	ϕ	a							
		a	b	ϕ	ϕ	b	ϕ	ϕ	a						
			a	b	ϕ	ϕ	b	ϕ	ϕ	a					
				a	b	ϕ	ϕ	b	ϕ	ϕ	a				
					a	b	ϕ	ϕ	b	ϕ	ϕ	a			
						a	b	ϕ	ϕ	b	ϕ	ϕ	a		
							a	b	ϕ	ϕ	b	ϕ	ϕ	a	
								a	b	ϕ	ϕ	b	ϕ	ϕ	a

Table 6.2: Illustration of a *block* for the pattern $ab\phi\phi b\phi\phi a$.

the text and pattern, patterns are designed that give bad performance if the algorithm is inspects characters in a fixed manner. Fixed algorithms consider the characters in each window of the text in a fixed order, an approach that is ubiquitous in string algorithms. In this section we consider non-fixed algorithms and derive an average-case lower bound for any algorithm solving the problem of wildcards in the pattern for with an arbitrary pattern. Clearly a lower bound for this problem also lower bounds the problem where wildcards appear in both the pattern and the text. When considering this problem the order characters are inspected become very important, below we give an example which illustrates the issues inspection schemes can cause.

Example 76. *Consider the block displayed in Table 6.2 were each row represents a occurrence of the pattern from each starting position and each column shows the ways in which each access could affect an occurrence of the pattern. Consider the situation after an inspection of positions* $4, 5$ *and* 6*. The candidate starting at position* 2 *and* 3 *have been intersected only once, that is it still has a relatively high probability of not being ruled out.*

As we can see in Example 76, some inspection schemes do not have much effect on the expected number of candidates not yet ruled out. It is the case that inspections that, should wildcards not be present, would lead to a large reduction in the expected number of candidates may give very little information when wildcards exist. This is what makes average-case algorithms for pattern matching with wildcards so sensitive to the inspection scheme.

Consider that we have a pattern of length m which contains $g < m$ wildcard characters and a text of length n. We partition the text into non-overlapping segments of size $2m$ which are referred to as *blocks*, and only consider that we have to report all matches or from within each block. This

is an optimistic assumption as this excludes those matches which overlap two blocks. In the following section we will determine a lower bound for the number of character inspections required for *one* block. The lower bound for the general problem can then be derived.

For each block we call all $0, \ldots, m-1$ possible starting positions *candidates* and when we inspect a character from the block we call this a *block access*. The candidates affected by a block access are *intersected* by it. Given a block access i_j to block b we can only rule out candidate c if $b_{i_j} \neq p_{i_j-c+1}$. For all non-wildcard positions intersected by a given block access i_j, there is a probability of at most $1/\sigma$ that the candidate will not be ruled out. For those candidates where this block access aligns with a wildcard there is probability 1 that it will not be ruled out. Now we outline a few of optimistic assumptions used in our analysis.

- Any access intersects all m candidates.
- Intersections are distributed uniformly across all candidates.

The affect of this is that $m - g$ candidates have a chance of being ruled out at every block access. After k block accesses in this model we have made $(m - g)k$ intersections and we assume that these are distributed uniformly across all m candidates. This is optimistic as the following inequality holds, where the first summation is the expected number of candidates not ruled out in the uniformly distributed scheme and the second is for any other:

$$\sum_{i=0}^{m-1} \frac{1}{\sigma^{\frac{(m-g)k}{m}}} \leq \sum_{i=0}^{m-1} \frac{1}{\sigma^{k_i}}$$

Where k_i represents the number of accesses to candidate i and $\sum_{i=0}^{m-1} k_i = (m - g)k$. Informally this means that we may only overestimate the number of candidates ruled out and our result is a lower bound. Clearly the left side of the summation can be evaluated as follows:

$$\sum_{i=0}^{m-1} \frac{1}{\sigma^{\frac{(m-g)k}{m}}} = \frac{m}{\sigma^{\frac{(m-g)k}{m}}}$$

For each candidate we need to either rule it out as a possible starting position or declare a match. So the optimal is to determine when we would expect to have ruled out every candidate position. We minimise the following so that

we expect to have at most one candidate left or until we have read all $2m$ positions.

$$\frac{m}{\sigma^{\frac{(m-g)k}{m}}} \leq 1$$

Rearranging this we get the following:

$$\log_\sigma m \leq \frac{(m-g)k}{m}$$

$$\frac{m \log_\sigma m}{m-g} \leq k$$

Now we know that the lower bound for each block is $\Omega(\frac{m \log_\sigma m}{m-g})$ and there are $n/2m$ windows. The result below follows:

Theorem 77. *The average-case lower bound for wildcard matching with wildcards only in the pattern is* $\Omega(\frac{n \log_\sigma m}{m-g})$.

For values of g such that $m-g = \Theta(m)$ this does not give us much additional insight as the lower bound matches exact string matching. However, consider the extreme cases such that $g = m - f$ where $\frac{\log_\sigma m}{2} \leq f$ we see that we must inspect the following number of characters:

$$\frac{m \log_\sigma m}{m-(m-f)} = \frac{m \log_\sigma m}{f} \leq k$$

For values of f less than $\frac{\log_\sigma m}{2}$ we must check all $2m$ characters in the block. So for $g = m - x \log_\sigma m$ the presented algorithm is optimal. Intuitively this is because as we increase the number of wildcards each block access gives us less information.

Finally we discuss a strategy for inspecting positions of a block and show that a greedy scheme performs an optimal number of character comparisons. By greedy we mean that at each step the block access which would most greatly reduce the expected number of remaining positions is chosen. For a candidate i let k_i be the number of times it has been accessed. Now for each position in a block we define the following set of candidates it affects. For each $0 \leq i < m$ let $\mathcal{B}_i$ be the set of candidates that the block access i intersects.

The effect on the expected number of candidates not ruled out by inspection some position ℓ is given by the following. Let $\mathcal{U} = \{0, 1, \ldots, m-1\} - \mathcal{B}_\ell$

and k_i denote the number of times candidate i has been intersected before the block access to ℓ.

$$\sum_{i=0}^{m-1} \frac{1}{\sigma^{k_i}} - \sum_{j \in \mathcal{U}} \frac{1}{\sigma^{k_j}} - \sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{k_h+1}}$$

The greedy inspection maximises the last two terms of the above summation at each step. The intuition behind this scheme is based on our proof of the lower bound presented above. In the proof we saw that evenly distributing accesses across all candidates minimises the expectation. The greedy scheme attempts to simulate this behaviour by picking the access which most minimises the probability at each step. We now show that this is in fact optimal.

Theorem 78. *The greedy inspection scheme performs an optimal number of character comparisons.*

Proof. The optimal number of character comparisons is achieved by a scheme minimised the number of inspections needed to expect that less than 1 candidate remains. We proceed by induction on the number of block access and claim that the greedy scheme is an optimal scheme. Let $g_0, g_1, .. g_{M-1}$ be the block access to candidates made by the greedy inspection scheme, we refer to this as the g scheme. The base case is simple, we pick the block access which minimises the expected number of candidates not ruled out, by definition this is the minimum.

Assume that for some i it is true that for the greedy scheme the number of expected candidates is the smallest possible after i accesses. Let $k_0, k_1, \ldots, k_i$ be an arbitrary inspection scheme, we refer to this as the k scheme. If the k scheme and g scheme are equal then we are done, otherwise we have a few cases to consider. Consider the case of $i+1$, each inspection scheme is required to pick a new access. Should they pick the same access ℓ then the we have a number of cases.

Let $\mathcal{B}_\ell$ be the set of candidates that the block access ℓ intersects. If the following holds:

$$\sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{k_h}} \leq \sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{g_h}}$$

Then so does the following:

$$\sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{k_h+1}} \leq \sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{g_h+1}}$$

Additionally, if the following is true:

$$\sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{k_h}} \geq \sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{g_h}}$$

Then so is the following:

$$\sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{k_h+1}} \geq \sum_{h \in \mathcal{B}_\ell} \frac{1}{\sigma^{g_h+1}}$$

Therefore, should they pick the same access the greedy scheme is still the minimum after $i+1$ accesses by the induction hypothesis.

Let $\mathcal{G}$ be the inspections made by the g scheme and $\mathcal{K}$ be the inspection made by the k scheme. Now let $\mathcal{C} = \{\{0, 1, \ldots 2m-1\} - \mathcal{G}\} - \mathcal{K}$. Should they not pick the same access then either the g scheme must still be optimal, or at least one scheme is not picking from $\mathcal{C}$. By definition any access in $\mathcal{C}$ is available to both schemes, if there existed an access that made the k scheme optimal then the g scheme can also pick it and remain optimal.

If they pick a different access then either k picks an access in $\mathcal{G}$ or g picks an access in $\mathcal{K}$. Should it be the case that the greedy scheme picks from $\mathcal{K}$ and the k scheme picks from $\mathcal{C}$ the k scheme must still be larger, if this was not the case then the greedy scheme could also pick the access the k scheme has picked and remain optimal.

The remaining case is that k picks from $\mathcal{G}$, we denote this access k_p. Let $k_u \in \mathcal{K}/\mathcal{G}$ and consider the inspection scheme $\{\mathcal{K}/k_u\} \cup k_p$, now let the g scheme pick access k_u at step $i+1$. By the induction hypothesis the $\mathcal{G}$ scheme is optimal for i accesses so $\mathcal{K} - k_p$ is larger or the same as the greedy scheme. Now we are in the situation where both schemes have the same access, k_u considered as the new access. By the above analysis on two schemes which pick the same accesses the greedy scheme remains optimal. □

6.4 Conclusions and Future Work

In this chapter we have investigated the average-case complexity of two wildcard matching problems by analysing the algorithms average-case and expected-case complexity. The original notion of average-case complexity in string matching is that of Knuth, however, for exact and approximate string matching the expected-case and average-case complexities are the same. Considering the expected-case complexity also gives us some insight into the

considered problems. Here we see that although there exists hard patterns which must take longer in the fixed model, but given a random pattern we do not expect it to be much harder than exact string matching in either model. This suggests that most patterns are actually easy to process on average. Although we have shown that the greedy inspection scheme is optimal, it is not clear if it matches the lower bound we have presented. Determining a tight bound on the average-case complexity of wildcard matching and designing an algorithm to implement this are future works.

It may seem that the time and space complexities of the dictionary are quite large. However, state of the art linear query time index of Bille *et al.* has a space complexity $\mathcal{O}(\sigma^{g^2} n \log^g \log n)$. For the expected-case space usage the exponent of m in both the space and time complexity is heavily dependent on the size of the alphabet when wildcards in the text are considered. For the DNA alphabet this is reduced to approximately $\mathcal{O}(m^{6.27} \log_{\frac{\sigma+1}{2}} m)$ for time and $\mathcal{O}(m^{5.27})$ for space, these values will tend to $\mathcal{O}(m^4 \log_{\frac{\sigma+1}{2}} m)$ and $\mathcal{O}(m^3)$ as the alphabet size increases.

7

Circular String Matching

A circular string of length n can be viewed as a traditional linear string which has the left- and right-most symbols wrapped around and stuck together in some way. Under this notion, the same circular string can be seen as n different linear strings, which would all be considered equivalent. Given a string x of length n, we denote by $x^i = x[i \,..\, n-1]x[0 \,..\, i-1]$, $0 < i < n$, the i-th *rotation* of x and $x^0 = x$. Consider, for instance, the string $x = x^0 = \texttt{abababbc}$; this string has the following rotations: $x^1 = \texttt{bababbca}$, $x^2 = \texttt{ababbcab}$, $x^3 = \texttt{babbcaba}$, $x^4 = \texttt{abbcabab}$, $x^5 = \texttt{bbcababa}$, $x^6 = \texttt{bcababab}$, $x^7 = \texttt{cababab}$.

Circular strings appear prominently in the context of DNA and RNA sequences. Circular structures occur in viruses, bacteria, eukaryotic cells, and archaea. In [57], it was noted that, due to this, algorithms on circular strings may be important in the analysis of organisms with such structure. For instance, circular strings have been studied before in the context of sequence alignment. In [96], basic algorithms for pairwise and multiple circular sequence alignment were presented. These results were later improved in [42], where an additional preprocessing stage was added to speed up the execution time of the algorithm. Later, in [83], the authors presented efficient algorithms for finding the optimal circular consensus sequence and alignment for certain cases.

Here we consider the problem of finding occurrences of a pattern x of length m with circular structure in a text t of length n with linear structure. For instance, the DNA sequence of many viruses has circular structure, so if a biologist wishes to find occurrences of a particular virus in a carriers DNA sequence—which may not be circular—they must consider how to locate all positions in t that at least one rotation of x occurs. This is the problem of *circular string matching*.

The problem of exact circular string matching has been considered in [89],

where an $\mathcal{O}(n)$-time algorithm was presented. A naive solution with quadratic complexity consists in applying a classical algorithm for searching a finite set of strings after having built the *trie* of rotations of x. The approach presented in [89] consists in preprocessing x by constructing a *suffix automaton* of the string xx, by noting that every rotation of x is a factor of xx. Then, by feeding t into the automaton, the lengths of the longest factors of xx occurring in t can be found by the links followed in the automaton in time $\mathcal{O}(n)$. The authors of [48] presented an optimal average-case algorithm for exact circular string matching, by also showing that the average-case lower bound for single string matching of $\mathcal{O}(n \log_\sigma m/m)$ also holds for circular string matching. Very recently, in [31], the authors presented two fast average-case algorithms based on word-level parallelism. The first algorithm requires average-case time $\mathcal{O}(n \log_\sigma m/w)$, where w is the number of bits in the computer word. The second one is based on a mixture of word-level parallelism and q-grams. The authors showed that with the addition of q-grams, and by setting $q = \mathcal{O}(\log_\sigma m)$, an optimal average-case time of $\mathcal{O}(n \log_\sigma m/m)$ is achieved. Indexing circular patterns [68] and variations of approximate circular string matching under the edit distance model [88]—both based on the construction of a *suffix tree*—have also been considered.

In this chapter, we consider the following problems.

Problem 79 (Exact Circular String Matching)**.** *Given a pattern x of length m and a text t of length $n > m$, find all factors u of t such that $u = x^i$, $0 \leq i < m$.*

Problem 80 (Approximate Circular String Matching with k-mismatches)**.** *Given a pattern x of length m, a text t of length $n > m$, and an integer threshold $k < m$, find all factors u of t such that $u \equiv_k^H x^i$, $0 \leq i < m$.*

Problem 81 (Approximate Circular String Matching with k-differences)**.** *Given a pattern x of length m, a text t of length $n > m$, and an integer threshold $k < m$, find all factors u of t such that $u \equiv_k^E x^i$, $0 \leq i < m$.*

The aforementioned algorithms for the exact case exhibit the following disadvantages: first, they cannot be applied in a biological context since both, single nucleotide polymorphisms, as well as errors introduced by wet-lab sequencing platforms might have occurred in the sequences; second, it is not clear whether they could easily be adapted to deal with the approximate case. Similar to the exact case [48], it can be shown that the average-case lower bound for single approximate string matching of $\mathcal{O}(n(k + \log_\sigma m)/m)$ [29] also holds for approximate circular string matching with k-mismatches under both Hamming and edit distance. To the best of our knowledge, no

optimal average-case algorithm exists for this problem. Therefore, one could use the optimal algorithm for multiple approximate string matching, presented in [50], for matching the $r = m$ rotations of x requiring, on average, time $\mathcal{O}(n(k + \log_\sigma rm)/m)$, only if $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$, $r = \mathcal{O}(\min(n^{1/3}/m^2, \sigma^{o(m)}))$, and we have $\mathcal{O}(m^4 r^2 \sigma^{\mathcal{O}(1)})$ space available; which is impractical for large m: e.g. the genome of the smallest known viruses replicating autonomously in eukaryotic cells is around 1.8KB long.

Our Contribution. We present a new suboptimal average-case algorithm for exact circular string matching requiring time and space $\mathcal{O}(n)$. Based on our novel solution for the exact case, we present a new fast average-case algorithm for approximate circular string matching with k-mismatches, under the Hamming distance model, requiring time $\mathcal{O}(n)$ for moderate values of k, that is $k = \mathcal{O}(m/\log_\sigma m)$, and space $\mathcal{O}(n)$. Finally, we present an average-case optimal algorithm for circular string matching under Hamming and edit distance which runs in average-case time $\mathcal{O}(\frac{n(k+\log_\sigma m)}{m})$ for $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$.

7.1 Properties of the Partitioning Technique

In this section, we give a brief outline of the *partitioning* technique in general; and then show some properties of the version of the technique we use for our algorithms. The partitioning technique, introduced in [131], and in some sense earlier in [112], is an algorithm based on *filtering out* candidate positions that could never give a solution to speed up string-matching algorithms. An important point to note about this technique is that it reduces the search space but does not, by design, verify potential occurrences. To create a string-matching algorithm filtering must be combined with some verification technique. The idea behind the partitioning technique was initially proposed for approximate string matching, but here we show that this can also be used for exact circular string matching.

The idea behind the partitioning technique is to partition the given pattern in such a way that at least one of the fragments must occur exactly in any valid approximate occurrence of the pattern. It is then possible to search for these fragments exactly to give a set of *candidate* occurrences of the pattern. It is then left to the verification portion of the algorithm to check if these are valid approximate occurrences of the pattern. It has been experimentally shown that this approach yields very good practical performance on large-scale datasets [52], even if it is not theoretically optimal.

For exact circular string matching, for an efficient solution, we cannot simply apply well-known exact string-matching algorithms, as we must also take into account the rotations of the pattern. We can, however, make use of the partitioning technique and, by choosing an appropriate number of fragments, ensure that at least one fragment must occur in any valid exact occurrence of a rotation. Lemma 83 together with the following fact provide this number.

Fact 82. *Any rotation of $x = x[0\,..\,m-1]$ is a factor of $x' = x[0\,..\,m-1]x[0\,..\,m-2]$; and any factor of length m of x' is a rotation of x.*

Lemma 83. *If we partition $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ in 4 fragments of length $\lfloor (2m-1)/4 \rfloor$ and $\lceil (2m-1)/4 \rceil$, at least one of the 4 fragments is a factor of any factor of length m of x'.*

Proof. Let ℓ_f denote the length of the fragment. If we partition x' in at least 4 fragments of length $\lfloor (2m-1)/4 \rfloor$ and $\lceil (2m-1)/4 \rceil$, we have that

$$\ell_f \leq (2m-1)/4,$$

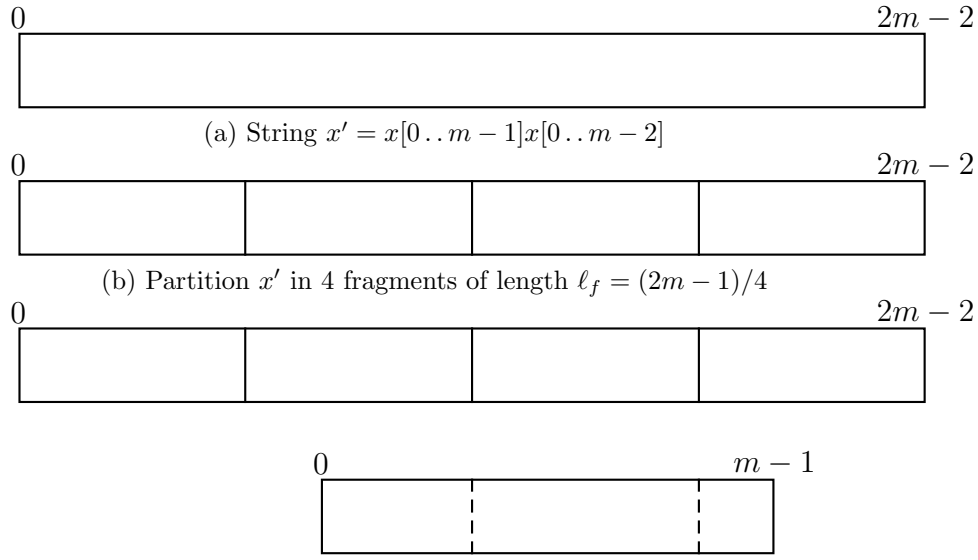
which gives $2m > 4\ell_f$ and $m > 2\ell_f$. Therefore any factor of length m of x', and, by Fact 82, any rotation of x, must contain at least one of the fragments. For a graphical illustration of this proof inspect Figure 7.1. □

Lemma 84. *Let x and $y = y_0 y_1 \ldots y_k$ be two strings, both of length n, such that $y_0, y_1, \ldots, y_k$ are $k+1 \leq n$ non-empty strings and $x \equiv_k y$. Then there exists at least one string y_i, $0 \leq i \leq k$, starting at position j of y, $0 \leq j < n$, occurring at position j of x.*

Proof. Immediate from the pigeonhole principle—if n items are put into $m < n$ pigeonholes, then at least one pigeonhole must contain more than one item. □

Based on Lemma 84, we take a similar approach to the one described by Lemma 83, to obtain the sufficient number of fragments in the case of approximate circular string matching with k-mismatches.

Lemma 85. *If we partition $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ in $2k+4$ fragments of length $\lfloor (2m-1)/(2k+4) \rfloor$ and $\lceil (2m-1)/(2k+4) \rceil$, at least $k+1$ of the $2k+4$ fragments are factors of any factor of length m of x'.*

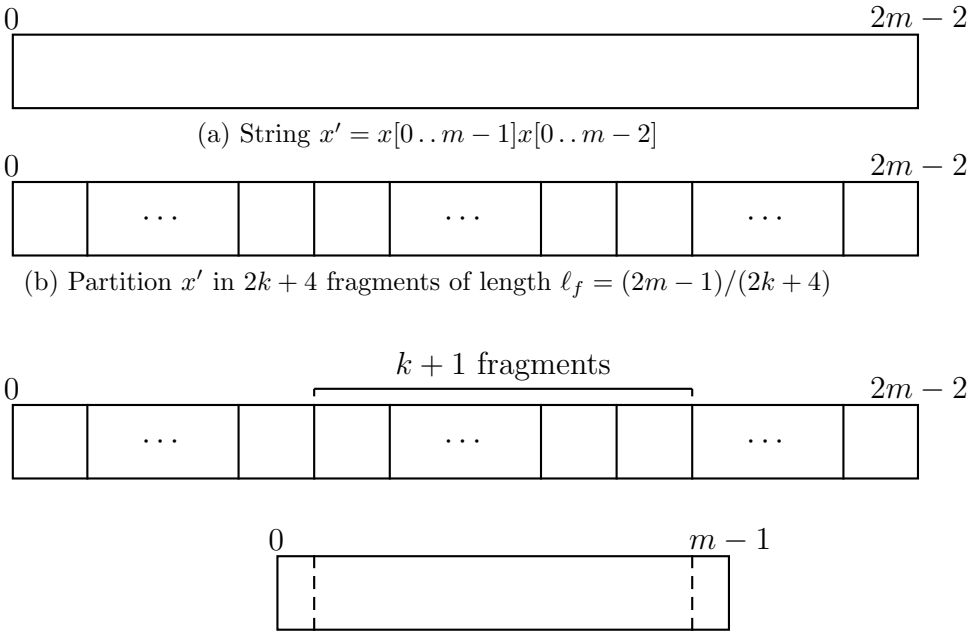


(a) String $x' = x[0\,..\,m-1]x[0\,..\,m-2]$

(b) Partition x' in 4 fragments of length $\ell_f = (2m-1)/4$

(c) $m > 2\ell_f$; any factor of length m of x' contains at least one of the 4 fragments

Figure 7.1: Illustration of Lemma 83



(a) String $x' = x[0\,..\,m-1]x[0\,..\,m-2]$

(b) Partition x' in $2k+4$ fragments of length $\ell_f = (2m-1)/(2k+4)$

(c) $m > (k+2)\ell_f$; any factor of length m of x' contains at least $k+1$ of the $2k+4$ fragments

Figure 7.2: Illustration of Lemma 85

Proof. Let ℓ_f denote the length of the fragment. If we partition x' in $2k+4$ fragments of length $\lfloor (2m-1)/(2k+4) \rfloor$ and $\lceil (2m-1)/(2k+4) \rceil$, we have that

$$\ell_f \leq (2m-1)/(2k+4),$$

which gives $2m-1 \geq 2(k+2)\ell_f$ and $m > (k+2)\ell_f$. Therefore any factor of length m of x', and, by Fact 82, any rotation of x, must contain at least $k+1$ of the fragments. For a graphical illustration of this proof inspect Figure 7.2. □

7.2 Exact Circular String Matching via Filtering

In this section, we present ECSMF, a new suboptimal average-case algorithm for exact circular string matching via filtering. It is based on the partitioning technique and a series of practical and well-established data structures such as the suffix array (for more details see [103]).

Longest Common Extension

First, we describe how to compute the longest common extension, denoted by lce, of two suffixes of a string in constant time (for more details see [63]). lce queries are an important part of the algorithms presented later on.

Let SA denote the array of positions of the sorted suffixes of string x of length n, i.e. for all $1 \leq r < n$, we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. Let $\mathsf{lcp}(r,s)$ denote the length of the longest common prefix of the strings $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$, for all $0 \leq r, s < n$, and 0 otherwise. Let LCP denote the array defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$, for all $1 < r < n$, and $\mathsf{LCP}[0] = 0$. We perform the following linear-time and linear-space preprocessing:

- compute arrays SA and iSA of x [103];
- compute array LCP of x [43];
- preprocess array LCP for range minimum queries, we denote this by $\mathsf{RMQ}_{\mathsf{LCP}}$ [44].

With the preprocessing complete, the lce of two suffixes of x starting at positions p and q can be computed in constant time in the following way [63]:

$$\mathsf{LCE}(x,p,q) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[p]+1, \mathsf{iSA}[q])].$$

Example 86. *Let the string* $x = \texttt{abbababba}$. *The following table illustrates the arrays* SA, iSA, *and* LCP *for* x.

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	b	b	a	b	a	b	b	a
SA$[i]$	8	3	5	0	7	2	4	6	1
iSA$[i]$	3	8	5	1	6	2	7	4	0
LCP$[i]$	0	1	2	4	0	2	3	1	3

We have $\mathsf{LCE}(x, 1, 2) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[2]{+}1, \mathsf{iSA}[1])] = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(6, 8)] =$ 1, *implying that the* lce *of* $\texttt{bbababba}$ *and* $\texttt{bababba}$ *is* 1.

Algorithm ECSMF

Given a pattern x of length m and a text t of length $n > m$, an outline of algorithm ECSMF for solving Problem 79 is as follows.

1. Construct the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ of length $2m-1$. By Fact 82, any rotation of x is a factor of x'.

2. The pattern x' is partitioned in 4 fragments of length $\lfloor (2m-1)/4 \rfloor$ and $\lceil (2m-1)/4 \rceil$. By Lemma 83, at least one of the 4 fragments is a factor of any rotation of x.

3. Match the 4 fragments against the text t using an Aho Corasick automaton [41]. Let $\mathcal{L}$ be a list of size Occ of tuples, where $< p_{x'}, \ell, p_t > \in \mathcal{L}$ is a 3-tuple such that $0 \leq p_{x'} < 2m-1$ is the position where the fragment occurs in x', ℓ is the length of the corresponding fragment, and $0 \leq p_t < n$ is the position where the fragment occurs in t.

4. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T = x't$. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T_r = \mathsf{rev}(tx')$, that is the reverse string of tx'.

5. For each tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we try to extend to the right via computing
$$\mathcal{E}_r \leftarrow \mathsf{LCE}(T, p_{x'} + \ell, 2m - 1 + p_t + \ell);$$
in other words, we compute the length $\mathcal{E}_r$ of the longest common prefix of $x'[p_{x'} + \ell\,..\,2m-1]$ and $t[p_t + \ell\,..\,n-1]$, both being suffixes of T. Similarly, we try to extend to the left via computing $\mathcal{E}_l$ using lce queries on the suffixes of T_r.

6. For each $\mathcal{E}_l, \mathcal{E}_r$ computed for tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we report all the valid starting positions in t by first checking if the total length $\mathcal{E}_l + \ell + \mathcal{E}_r \geq m$; that is the length of the full extension of the fragment is greater than or equal to m, matching at least one rotation of x. If that is the case, then we report positions

$$\max\{p_t - \mathcal{E}_\ell, p_t + \ell - m\}, \ldots, \min\{p_t + \ell - m + \mathcal{E}_r, p_t\}.$$

Example 87. *Let the pattern $x =$ `GGGTCTA` of length $m = 7$, and the text $t =$ `GATACGATACCTAGGGTGATAGAATAG`. Then $x' =$ `GGGTCTAGGGTCT` (Step 1). x' is partitioned in `GGGT`, `CTA`, `GGG`, and `TCT` (Step 2). Consider $< 4, 3, 10 > \in \mathcal{L}$, that is, fragment $x'[4\,..\,6] =$ `CTA`, of length $\ell = 3$, occurs at starting position $p_t = 10$ in t (Step 3). So $T =$ `GGGTCTAGGGTCTGATACGATACCTAGGGTGATAGAATAG` and $T_r =$ `TCTGGGATCTGGGGATAAGATAGTGGGATCCATAGCATAG` (Step 4). Extending to the left gives $\mathcal{E}_l = 0$, since $T_r[9] \neq T_r[30]$; and extending to the right gives $\mathcal{E}_r = 4$, since $T[7\,..\,10] = T[26\,..\,29]$ and $T[11] \neq T[30]$ (Step 5). We check that $\mathcal{E}_l + \ell + \mathcal{E}_r = 7 = m$, and therefore we report position 10 (Step 6):*

$$p_t - \mathcal{E}_\ell = 10 - 0 = 10, \ldots, p_t + \ell - m + \mathcal{E}_r = 10 + 3 - 7 + 4 = 10;$$

that is, $x^4 =$ `CTAGGGT` occurs at starting position 10 in t.

Theorem 88. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, and a text t of length $n > m$ drawn from Σ, algorithm ECSMF requires average-case time $\mathcal{O}(n)$ to solve Problem 79.*

Proof. Constructing and partitioning the string x' from x can trivially be done in time $\mathcal{O}(m)$ (Step 1-2). Building the Aho-Corasick automaton of the 4 fragments requires time $\mathcal{O}(m)$; and the search time is $\mathcal{O}(n + Occ)$ (Step 3) [41]. The preprocessing step for the lce queries on the suffixes of T and T_r can be done in time $\mathcal{O}(n)$ (Step 4). Computing $\mathcal{E}_l$ and $\mathcal{E}_r$ for each occurrence of a fragment requires time $\mathcal{O}(Occ)$ (Step 5). For each extended occurrence of a fragment, we report $\mathcal{O}(m)$ valid starting positions, thus $\mathcal{O}(mOcc)$ in total (Step 6). Since the expected number Occ of occurrences of the 4 fragments in t is $4n/\sigma^{(2m-1)/4} = \mathcal{O}(\frac{n}{\sigma^{\frac{2m-1}{4}}})$, algorithm ECSMF requires average-case time $\mathcal{O}((1 + \frac{m}{\sigma^{\frac{2m-1}{4}}})n)$. It achieves average-case time $\mathcal{O}(n)$ *iff*

$$f = \frac{4m}{\sigma^{\frac{2m-1}{4}}} n \leq cn$$

for some fixed constant c. For $\sigma = 2$, the maximum value of f is attained at

$$m = 2/\ln 2 \approx 2.8853$$

and so for $\sigma > 1$ we get

$$\frac{4m}{\sigma^{\frac{2m-1}{4}}}n \leq 5.05n.$$

□

7.3 Approximate Circular String Matching with k-mismatches via Filtering

In this section, based on the ideas presented in algorithm ECSMF, we present algorithms ACSMF and ACSMF-Simple, two new fast average-case algorithms for approximate circular string matching with k-mismatches via filtering.

Algorithm ACSMF

The first four steps of algorithm ACSMF are essentially the same as in algorithm ECSMF. A small difference exists in Step 2, where the sufficient number of fragments in the case of approximate circular string matching with k-mismatches is used. The main difference is in Step 5, where algorithm ACSMF tries to extend $k+1$ times to the right and $k+1$ times to the left. Given a pattern x of length m, a text t of length $n > m$, and an integer threshold $k < m$, an outline of algorithm ACSMF for solving Problem 81 is as follows.

1. Construct the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ of length $2m-1$. By Fact 82, any rotation of x is a factor of x'.

2. The pattern x' is partitioned in $2k+4$ fragments of length $\lfloor (2m-1)/(2k+4) \rfloor$ and $\lceil (2m-1)/(2k+4) \rceil$. By Lemma 85, at least $k+1$ of the $2k+4$ fragments are factors of any rotation of x.

3. Match the $2k+4$ fragments against the text t using an Aho Corasick automaton [41]. Let $\mathcal{L}$ be a list of size Occ of tuples, where $< p_{x'}, \ell, p_t > \in \mathcal{L}$ is a 3-tuple such that $0 \leq p_{x'} < 2m-1$ is the position where the fragment occurs in x', ℓ is the length of the corresponding fragment, and $0 \leq p_t < n$ is the position where the fragment occurs in t.

4. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T = x't$. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T_r = \mathsf{rev}(tx')$, that is the reverse string of tx'.

5. For each tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we try to extend $k+1$ times to the right via computing

$$\mathcal{E}_r^0 \leftarrow \mathsf{LCE}(T, p_{x'} + \ell, 2m - 1 + p_t + \ell) + 1$$

$$\mathcal{E}_r^1 \leftarrow \mathsf{LCE}(T, p_{x'} + \ell + \mathcal{E}_r^0, 2m - 1 + p_t + \ell + \mathcal{E}_r^0) + 1$$

$$\ldots$$

$$\mathcal{E}_r^{k-1} \leftarrow \mathsf{LCE}(T, p_{x'} + \ell + \mathcal{E}_r^{k-2}, 2m - 1 + p_t + \ell + \mathcal{E}_r^{k-2}) + 1$$

$$\mathcal{E}_r^k \leftarrow \mathsf{LCE}(T, p_{x'} + \ell + \mathcal{E}_r^{k-1}, 2m - 1 + p_t + \ell + \mathcal{E}_r^{k-1});$$

in other words, we compute the length $\mathcal{E}_r^k$ of the longest common prefix of $x'[p_{x'} + \ell \,..\, 2m-1]$ and $t[p_t + \ell \,..\, n-1]$, both being suffixes of T, with k mismatches. Similarly, we try to extend to the left $k+1$ times via computing $\mathcal{E}_l^k$ using lce queries on the suffixes of T_r.

6. For each tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$ we try to extend, we also maintain an array M of size $2m-1$, initialised with zeros, where we mark the position of the i-th left and right mismatch, $1 \leq i \leq k$, by setting

$$\mathsf{M}[p_{x'} - \mathcal{E}_l^{i-1} - 1] \leftarrow 1 \text{ and } \mathsf{M}[p_{x'} + \ell + \mathcal{E}_r^{i-1}] \leftarrow 1.$$

7. For each $\mathcal{E}_l^k, \mathcal{E}_r^k, \mathsf{M}$ computed for tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we report all the valid starting positions in t by first checking if the total length $\mathcal{E}_l^k + \ell + \mathcal{E}_r^k \geq m$; that is the length of the full extension of the fragment is greater than or equal to m. If that is the case, then we count the total number of mismatches of the occurrences at starting positions

$$\max\{p_t - \mathcal{E}_\ell^k, p_t + \ell - m\}, \ldots, \min\{p_t + \ell - m + \mathcal{E}_r^k, p_t\},$$

by first summing up the mismatches for the leftmost starting position

$$\mu_j \leftarrow \mathsf{M}[p_{x'} - \mathcal{E}_l^k] + \ldots + \mathsf{M}[p_{x'} - \mathcal{E}_l^k + m - 1], \text{ where } j = \max\{p_t - \mathcal{E}_\ell^k, p_t + \ell - m\}.$$

For each subsequent position $j+1$, we subtract the value of the leftmost element of M computed for μ_j and add the value of the next element to compute μ_{j+1}. In case $\mu_j \leq k$, we report position j.

Example 89. *Let the pattern $x =$ `GGGTCTA` of length $m = 7$, the text $t =$ `GATACGATACCTAGGGTGATAGAATAG`, and $k = 1$. Then $x' =$ `GGGTCTAGGGTCT` (Step 1). x' is partitioned in `GGG`, `TC`, `TA`, `GG`, `GT`, and `CT` (Step 2). Consider $< 9, 2, 15 > \in \mathcal{L}$, that is, fragment $x'[9 \,..\, 10] =$ `GT`, of length $\ell = 2$, occurs at starting position $p_t = 15$ in t (Step 3).*

Then $T =$ `GGGTCTAGGGTCTGATACGATACCTAGGGTGATAGAATAG`
and $T_r =$ `TCTGGGATCTGGGGATAAGATAGTGGGATCCATAGCATAG` *(Step 4). Extending to the left gives* $\mathcal{E}_l^k = 6$, *since* $T_r[4\,..\,9] \equiv_k T_r[25\,..\,30]$ *and* $T_r[10] \neq T_r[31]$; *and extending to the right gives* $\mathcal{E}_r^k = 1$, *since* $T[11] \equiv_k T[30]$ *and* $T[12] \neq T[31]$ *(Step 5). We also set* $\mathsf{M}[3] = 1$ *and* $\mathsf{M}[11] = 1$ *(Step 6). We check that* $\mathcal{E}_l + \ell + \mathcal{E}_r = 9 > m$, *and therefore we report positions 10, since* $\sum_{i=4}^{10} \mathsf{M}[i] = 0 < k$, *and 11, since* $\sum_{i=5}^{11} \mathsf{M}[i] = 1 = k$ *(Step 7):*

$$p_t + \ell - m = 15 + 2 - 7 = 10, \ldots, p_t + \ell - m + \mathcal{E}_r = 15 + 2 - 7 + 1 = 11;$$

that is, $x^4 =$ `CTAGGGT` *and* $x^5 =$ `TAGGGTC` *occur at starting position* 10 *in* t *with no mismatch and at starting position* 11 *in* t *with 1 mismatch, respectively.*

Theorem 90. *Given a pattern* x *of length* m *drawn from alphabet* Σ, $\sigma = |\Sigma|$, *a text* t *of length* $n > m$ *drawn from* Σ, *and an integer threshold* $k < m$, *algorithm* ACSMF *requires average-case time* $\mathcal{O}((1+\frac{km}{\sigma^{\frac{2m-1}{2k+4}}})n)$ *and space* $\mathcal{O}(n)$ *to solve Problem 81.*

Proof. Constructing and partitioning the string x' from x can trivially be done in time $\mathcal{O}(m)$ (Step 1-2). Building the Aho-Corasick automaton of the $2k + 4$ fragments requires time $\mathcal{O}(m)$; and the search time is $\mathcal{O}(n + Occ)$ (Step 3) [41]. The preprocessing step for the lce queries on the suffixes of T and T_r can be done in time and space $\mathcal{O}(n)$ (Step 4)—see Section 7.2. Computing $\mathcal{E}_l^k$ and $\mathcal{E}_r^k$ for each occurrence of a fragment requires time $\mathcal{O}(kOcc)$ (Step 5)—see Section 7.2. Maintaining array M is of no extra cost (Step 6). For each extended occurrence of a fragment, we report $\mathcal{O}(m)$ valid starting positions, thus $\mathcal{O}(mOcc)$ in total (Step 7). Since the expected number Occ of occurrences of the $2k+4$ fragments is $(2k+4)n/\sigma^{(2m-1)/(2k+4)} = \mathcal{O}(\frac{kn}{\sigma^{\frac{2m-1}{2k+4}}})$, algorithm ACSMF requires average-case time $\mathcal{O}((1+\frac{km}{\sigma^{\frac{2m-1}{2k+4}}})n)$ and space $\mathcal{O}(n)$. □

Corollary 91. *Given a pattern* x *of length* m *drawn from alphabet* Σ, $\sigma = |\Sigma|$, *a text* t *of length* $n > m$ *drawn from* Σ, *and an integer threshold* $k = \mathcal{O}(m/\log_\sigma m)$, *algorithm* ACSMF *requires average-case time* $\mathcal{O}(n)$.

Proof. Algorithm ACSMF achieves average-case time $\mathcal{O}(n)$ *iff*

$$m(2k + 4)n/\sigma^{(2m-1)/(2k+4)} \leq cn$$

for some fixed constant c. Let $r = (2m - 1)/(2k + 4)$. We have

$$m(2k + 4)n/\sigma^r \leq cn.$$

Since $k < m$, we can (pessimistically) replace k by $m - 1$. Then we have

$$2m(m+1)n/\sigma^r \leq cn.$$

Solving for r, and using $k \leq (2m-1)/2r - 2$, gives the maximum value of k, that is

$$k = \mathcal{O}(m/\log_\sigma m).$$

□

Algorithm ACSMF-Simple

Algorithm ACSMF-Simple is very similar to Algorithm ACSMF. The only differences are:

- Algorithm ACSMF-Simple does not perform Step 4 of Algorithm ACSMF;
- For each tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, Step 5 of Algorithm ACSMF is performed without the use of the pre-computed indexes. In other words, we compute $\mathcal{E}_r^k$ and $\mathcal{E}_\ell^k$ by simply performing letter comparisons and counting the number of mismatches occurred. The extension stops right before the $k+1$th mismatch.

Fact 92. *The expected number of letter comparisons required for each extension in algorithm ACSMF-Simple is less than* 3.

Proof. Recall that on an alphabet of size σ, the probability that two random strings of length ℓ are equal is $(1/\sigma)^\ell$. Thus, given two long strings, and setting $r = 1/\sigma$, there is probability r that the initial letters are equal, r^2 that the prefixes of length two are equal, and so on. Thus the expected number of positions to be matched before inequality occurs is

$$S = r + 2r^2 + \cdots + (n-1)r^{n-1},$$

for some $n \geq 2$. Hall & Knight [58, p. 44] tell us that

$$S = r(1 - r^{n-1})/(1-r)^2 - (n-1)r^n/(1-r),$$

which as $n \to \infty$ approaches $r/(1-r)^2 < 2$ for all r. Thus S, the expected number of matching positions, is less than 2, and hence the expected number of letter comparisons required for each extension in algorithm ACSMF-Simple is less than 3. □

Theorem 93. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, a text t of length $n > m$ drawn from Σ, and an integer threshold $k < m$, algorithm ACSMF-Simple requires average-case time $\mathcal{O}((1 + \frac{km}{\sigma^{\frac{2m-1}{2k+4}}})n)$ and space $\mathcal{O}(m)$ to solve Problem 81.*

Proof. By Fact 92, computing $\mathcal{E}_\ell^k$ and $\mathcal{E}_r^k$ for each occurrence of a fragment requires time $\mathcal{O}(k\,Occ)$. Therefore algorithm ACSMF-Simple requires average-case time $\mathcal{O}((1+\frac{km}{\sigma^{\frac{2m-1}{2k+4}}})n)$. The required space is reduced to $\mathcal{O}(m)$ since Step 4 of Algorithm ACSMF is not performed. □

Corollary 94. *Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, a text t of length $n > m$ drawn from Σ, and an integer threshold $k = \mathcal{O}(m/\log_\sigma m)$, algorithm ACSMF-Simple requires average-case time $\mathcal{O}(n)$.*

In practical cases, algorithm ACSMF-Simple should be preferred over algorithm ACSMF as (i) it has less memory requirements (see Theorem 93); and (ii) it avoids the construction of a series of data structures (see Section 7.2 in this regard).

7.4 Edit Distance Model

Algorithm ACSMF-Simple could be easily extended for approximate circular string matching under the *edit distance* model (for a definition, see [39]). Since each single-letter edit operation can change at most one of the $2k + 4$ fragments of x', any set of at most k edit operations leaves at least one of the fragments untouched. In other words, Lemma 84 holds under the edit distance model as well [54]. An area of length $\mathcal{O}(m)$ surrounding each potential occurrence found in the filtration phase (Steps 1-3 of algorithm ACSMF) is then searched using the standard dynamic-programming algorithm in time $\mathcal{O}(m^2)$ [129] and space $\mathcal{O}(m)$ [59]. Since the expected number Occ of occurrences of the $2k+4$ fragments is $\mathcal{O}(\frac{kn}{\sigma^{\frac{2m-1}{2k+4}}})$, the average-case time complexity becomes $\mathcal{O}((1 + \frac{km^2}{\sigma^{\frac{2m-1}{2k+4}}})n)$ and the space complexity remains $\mathcal{O}(m)$. When $k = \mathcal{O}(m/\log_\sigma m)$, the average-case time complexity is $\mathcal{O}(n)$.

7.5 Experimental Results

We implemented algorithms ACSMF and ACSMF-Simple as library functions to perform approximate circular string matching with k-mismatches. The

functions were implemented in the C programming language and developed under GNU/Linux operating system. They take as input arguments the pattern x of length m, the text t of length n, and the integer threshold $k < m$; and then return the list of starting positions of the occurrences of the rotations of x in t with k-mismatches as output. The library implementation is distributed under the GNU General Public License (GPL), and it is available at `http://www.inf.kcl.ac.uk/research/projects/asmf/`, which is set up for maintaining the source code and the man-page documentation. The experiments were conducted on a Desktop PC using one core of Intel i7 2600 CPU at 3.4 GHz under GNU/Linux.

Approximate circular string matching is a rather undeveloped area. To the best of our knowledge, there does not exist an optimal (average- or worst-case) algorithm for approximate circular string matching with k-mismatches. Therefore, keeping in mind that we wish to evaluate the efficiency of our algorithms in practical terms, we compared their performance to the respective performance of the C implementation[1] of the optimal average-case algorithm for multiple approximate string matching, presented in [50], for matching the $r = m$ rotations of x. We denote this algorithm by FredNava.

Tables 7.1 -7.3 illustrate elapsed-time and speed-up comparisons for various pattern sizes and moderate values of k, using a corpus of DNA data taken from the Pizza&Chili website [108]. As it is demonstrated by the experimental results, algorithm ACSMF-Simple is in all cases the fastest with a speed-up improvement of more than three orders of magnitude over FredNava. ACSMF is always the second fastest, while ACSMF-Simple still retains a speed-up improvement of more than one order of magnitude over ACSMF. Another important observation, also suggested by Corollaries 91 and 94, is that the ACSMF-based algorithms are essentially *independent* of m for moderate values of k.

So far in this chapter, we presented new average-case algorithms for exact and approximate circular string matching. Algorithm ECSMF for exact circular string matching requires average-case time $\mathcal{O}(n)$; and Algorithms ACSMF and ACSMF-Simple for approximate circular string matching with k-mismatches require time $\mathcal{O}(n)$ for moderate values of k, that is $k = \mathcal{O}(m/\log_\sigma m)$. We showed how the same results can be easily obtained under the edit distance model. The presented algorithms were also implemented as library functions. Experimental results demonstrate that the functions provided in this library accelerate the computations by more than three orders of magnitude compared to a naïve approach.

[1]Personal communication with author

		Elapsed Time (s)			Speed-up of ACSMF-Simple	
m	k	FredNava	ACSMF	ACSMF-Simple	FredNava	ACSMF
100	5	1.63	0.40	0.06	27	7
200	5	6.77	0.40	0.05	135	8
300	5	16.84	0.41	0.05	337	8
400	5	31.99	0.41	0.05	640	8
500	5	53.26	0.41	0.05	1065	8
600	5	81.35	0.41	0.05	1627	8
700	5	116.24	0.41	0.05	2325	8
800	5	158.73	0.41	0.06	2645	7
900	5	206.43	0.42	0.06	3440	7
1000	5	264.84	0.41	0.06	4414	7
100	10	1.65	0.43	0.05	33	9
200	10	6.94	0.40	0.05	139	8
300	10	16.55	0.41	0.05	331	8
400	10	31.70	0.40	0.05	634	8
500	10	53.11	0.41	0.05	1062	8
600	10	81.04	0.40	0.05	1620	8
700	10	116.25	0.41	0.06	1937	7
800	10	158.1	0.41	0.06	2635	7
900	10	207.33	0.41	0.05	4146	8
1000	10	264.11	0.41	0.05	5282	8
100	15	1.65	0.42	0.06	28	7
200	15	6.91	0.41	0.06	115	7
300	15	16.45	0.41	0.06	274	7
400	15	31.48	0.41	0.05	630	8
500	15	52.55	0.41	0.05	1051	8
600	15	80.46	0.41	0.05	1069	8
700	15	115.86	0.41	0.06	1931	7
800	15	157.81	0.41	0.06	2630	7
900	15	206.56	0.42	0.06	3443	7
1000	15	262.16	0.42	0.06	4369	7

Table 7.1: Elapsed-time and speed-up comparisons of algorithms ACSMF and ACSMF-Simple using DNA data for n = 10MB.

		Elapsed Time (s)		**Speed-up of ACSMF-Simple**
m	k	ACSMF	ACSMF-Simple	ACSMF
10000	100	6.54	0.67	10
11000	100	6.69	0.70	10
12000	100	6.57	0.72	9
13000	100	6.64	0.74	9
14000	100	6.58	0.75	9
10000	300	6.54	0.69	9
11000	300	6.67	0.69	10
12000	300	6.64	0.68	10
13000	300	6.71	0.71	9
14000	300	6.63	0.72	9
10000	500	6.74	0.66	10
11000	500	6.58	0.67	10
12000	500	6.69	0.66	10
13000	500	6.66	0.67	10
14000	500	6.71	0.68	10

Table 7.2: Elapsed-time and speed-up comparisons of algorithms ACSMF and ACSMF-Simple using DNA data for $n = 10$MB

		Elapsed Time (s)		**Speed-up of ACSMF-Simple**
m	k	ACSMF	ACSMF-Simple	ACSMF
50000	500	45.71	4.33	11
51000	500	45.81	4.35	11
52000	500	45.73	4.37	10
53000	500	44.99	4.40	10
54000	500	45.05	4.40	10
50000	700	45.00	4.26	11
51000	700	44.79	4.18	11
52000	700	44.96	4.36	10
53000	700	44.83	4.32	10
54000	700	45.00	4.32	10
50000	900	46.79	4.32	11
51000	900	44.89	4.28	10
52000	900	45.06	4.33	10
53000	900	45.14	4.35	10
54000	900	44.81	4.12	11

Table 7.3: Elapsed-time and speed-up comparisons of algorithms ACSMF and ACSMF-Simple using DNA data for $n = 50$MB.

In the rest of this chapter, we present a new average-case optimal algorithm for approximate circular string matching that reduces the preprocessing time and space requirements compared to previous average-case optimal algorithms. Additionally, we believe, the presented algorithm is conceptually more direct than the existing algorithms as it does not rely on the reduction to multiple approximate string matching.

7.6 Optimal Average-case Circular String Matching

In this section, we present our algorithm for approximate circular string matching under the edit distance model. The presented algorithm can be seen as consisting of two distinct schemes: the *searching* scheme which determines if the currently considered text window potentially has a valid occurrence; in case the window *may* contain a valid occurrence, we are required to check the window for valid occurrences of the pattern or any of its rotations; this is done through the *verification* scheme.

Intuitively, the algorithm considers a *sliding window* of length $m - k$ of the text, and reads enough q-grams such that it is likely to have found enough *differences* to skip the entire window. That is, we wish to make the probability of a verification being triggered sufficiently unlikely whilst ensuring we can shift the window a reasonable length.

The rest of this section is structured as follows. We first present an efficient incremental string comparison technique which forms the basis of the verification scheme. We then present the searching scheme of our algorithm which requires a preprocessing step. In fact, this preprocessing step is similar to the verification scheme. Finally, we show how plugging these schemes together results in a new average-case optimal algorithm for approximate circular string matching.

7.6.1 Verification Scheme

The verification scheme of our algorithm is based on incremental string comparison techniques. First we give an introduction to these techniques; and then explain how we use them in the verification scheme. The incremental string comparison problem was introduced by Landau *et al.* in [79]. The authors considered the following problem: given the edit distance between two strings A and B, how can the edit distance between A and bB or Bb be

efficiently derived, where b is an additional letter. Given a threshold on the number of differences k, they solve this problem and allow prepending and appending of letters in time $\mathcal{O}(k)$ per operation. Later the authors of [61] considered a generalisation of this problem with the aim of computing all maximal gapped palindromes in a string. The problem considered is a generalisation of the incremental string comparison problem considered in [79] as it considers how to efficiently derive the edit distance when prefixes are deleted and letters are prepended to A or B. The solution proposed in [61] also has a time complexity of $\mathcal{O}(k)$ per operation. The solution for the generalised incremental string comparison problem forms the basis of our verification step. The technique lends itself more naturally to circular string matching due to the increased flexibility it provides. We begin by recalling some of the main results from [61] required for our algorithm.

The main idea in both [79] and [61] is the efficient computation of the so-called *h-waves*. In the standard dynamic programming matrix, we say that a cell $\mathsf{D}[i, j]$ is on the diagonal d *iff* $j - i = d$. For each diagonal, we may have a lowest cell with value h; if $\mathsf{D}[i, j] = h$ and $\mathsf{D}[i+1, j+1] = h+1$ then $\mathsf{D}[i, j]$ is this cell for diagonal $j - i$. The h-wave, for all $0 \leq h \leq k$, is the position of all these cells across all diagonals, that is, a list H_h of length $\mathcal{O}(k)$, where each entry is a pair (i, j) such that $\mathsf{D}[i, j] = h$ and $\mathsf{D}[i+1, j+1] = h+1$. Note that the i-th wave can only contain entries on diagonal zero and the i diagonals either side of it, so for $0 \leq i \leq k$ every wave has size $\mathcal{O}(k)$. Both incremental string comparison techniques show some bounds on the possible values of the cells on h-waves and how to efficiently compute them. These h-waves define the entire dynamic programming matrix due to the monotonicity properties of the matrix. For any diagonal d, if we know the position of the lowest cell on d with value h and $h+1$, then we also know the value of every cell between these two cells: it must be $h + 1$. So given the h-waves of the matrix, for all $0 \leq h \leq k$, we have all the information that is in the standard dynamic programming matrix. The key result from our perspective is the following.

Let $\mathsf{cat}(u', u)$ denote the string obtained by concatenating string u' and string u. Let $\mathsf{del}(\alpha, u)$ denote the string obtained by deleting the prefix of length α of string u. Further let D' denote the standard dynamic programming matrix of $\mathsf{cat}(\mathsf{A}', \mathsf{A})$ and $\mathsf{del}(t_2, \mathsf{B})$, where $|\mathsf{A}'| = t_1$.

Theorem 95 ([61])**.** *The 0-wave, 1-wave, ... , and k-wave of matrix D' can be computed in time $\mathcal{O}((t_1 + t_2)k)$.*

If a window of the text triggers a verification then we have a window of length $m - k$ such that there exist some q-grams of the window that occur in x or its rotations with at most k differences in total. When we verify a

window, we check for occurrences of pattern x starting at every position in the window. For each position, we may have a factor of length at most $m+k$ representing an occurrence, meaning we must consider a factor w of the text of length $2m$ which we refer to as a *block*. This ensures we avoid missing any occurrences at the $m-k$ starting positions as $(m-k)+(m+k)=2m$.

For each possible starting position i, $0 \leq i < m-k$, we compute the 0-wave, 1-wave, ... , and k-wave for x and $w' = w[i\,..\,2m-1]$, the suffix of w starting at position i. To check if we have an occurrence, we must check the k-wave H_k. We iterate through each entry in the k-wave H_k; and if H_k has missing entries or contains entries on the last row of the matrix, then x occurs in w with at most k differences. If any diagonal has no entry on the k-wave then that diagonal reached the last row of the matrix with less than k differences; this means x occurs in w with less than k differences. Similarly we can check for the occurrences of the rotations of x using the incremental string comparison techniques.

We are now ready to outline the verification scheme, denoted by function VER. Given the pattern x of length m, an integer threshold $k < m$, and a block w of length $2m$ of the text t, function VER finds all factors u of w such that $u \equiv_k^E x^i$, $0 \leq i < m$.

Lemma 96. *Given the pattern x of length m, an integer threshold $k < m$, and string w of length $2m$, function VER requires time $\mathcal{O}(m^2k)$.*

Proof. Computing the edit distance between x and $w[0\,..\,2m-1]$ with at most k differences requires time $\mathcal{O}(mk)$ using the standard dynamic programming algorithm. By Theorem 95, computing the edit distance between all the rotations of the pattern and $w[i\,..\,2m-1]$ for a single position in w requires time $\mathcal{O}(mk)$; and there are $\mathcal{O}(m)$ positions in w. In total, this requires time $\mathcal{O}(mk+m^2k)$, that is $\mathcal{O}(m^2k)$. □

7.6.2 Searching Scheme

The searching scheme of the presented algorithm requires the preprocessing and indexing of the pattern x. We first present the preprocessing required and then present the searching technique itself.

Preprocessing.

We build a q-gram index in a similar way as the index proposed by Chang and Marr in [29]. Intuitively, we wish to determine the minimum possible edit

ALGORITHM VER$(x, m, k, w, 2m)$
{Where x is a string, m is the length of x, k is the allowed differences and w is a string of length $2m$. }
Compute the edit distance between x and $w' = w[0\,..\,2m-1]$ with at most k differences using the standard dynamic programming algorithm
Check for any occurrences using D, and if found, **report** an occurrence at position 0
for each $i \in \{1, m-k-1\}$ **do**
  for each $j \in \{1, m\}$ **do**
    Construct rotation x^j of x by removing the first letter of x^{j-1} and appending it to the end of x^{j-1}
    Compute the edit distance between x^j and $w' = w[i\,..\,2m-1]$ using the incremental string comparison techniques
    Check for any occurrences using H_k, and if found, **report** an occurrence at the current position i being checked
  end for
end for

distance between every q-gram and any factor of x or its rotations. Equivalently we find the minimum possible edit distance between every q-gram and any *prefix* of a factor of length $2q$ of x or its rotations. An index built like this allows us to determine a lower bound on the edit distance between some window of the text and x or its rotations without computing the edit distance between the window and x and each rotation separately. To build this index, we generate every string of length q on Σ, and find the minimum edit distance between it and all prefixes of factors of length $2q$ of x or its rotations. This information can easily be stored by generating a numerical representation of the q-gram and storing the minimum edit distance in an array at this location. If we know the numerical representation, we can then look up any entry in constant time. We determine the edit distance using the following preprocessing scheme, denoted by function PRE, which is similar to the verification scheme (function VER). Given the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ of length $2m-1$, function PRE finds the minimum edit distance between every q-gram on Σ and any factor u of length $2q$ of x'.

Lemma 97. *Given the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ of length $2m-1$ on Σ, $\sigma = |\Sigma|$, and $q < m$, function PRE requires time $\mathcal{O}(\sigma^q mq)$ and space $\mathcal{O}(\sigma^q)$.*

Proof. The time required for initialising array M is $\mathcal{O}(\sigma^q)$. The time required for computing the edit distance between $x'[0\,..\,2q-1]$ and s is $\mathcal{O}(q^2)$ using the

standard dynamic programming algorithm. By Theorem 95, computing the edit distance between all $2q$-grams of x' and s requires time $\mathcal{O}(mq)$. There exist $\mathcal{O}(\sigma^q)$ possible q-grams on Σ and so, in total, the time complexity is $\mathcal{O}(\sigma^q mq)$. Keeping array M in memory requires space $\mathcal{O}(\sigma^q)$. □

ALGORITHM $\mathsf{PRE}(x', 2m-1, q)$
{Where x' is a string, $2m-1$ is the length of x and q is an integer. }
$\mathsf{M}[0\,..\,\sigma^q - 1] \leftarrow 0$
$j \leftarrow 0$
for each $s \in \Sigma^q$ **do**
  Compute the edit distance between $u = x'[0\,..\,2q-1]$ and s using the standard dynamic programming algorithm. Set $E_{\min}$ equal to the minimum edit distance between s and any prefix of u using D
  for each $i \in \{1, 2m-q-1\}$ **do**
    $u \leftarrow u[1\,..\,2q-1]x'[2q-1+i]$
    Compute the edit distance E' between u and s using the incremental string comparison techniques. Set E' equal to the minimum edit distance between s and any prefix of u using H_q
    if $E' < E_{\min}$ **then**
      $E_{\min} \leftarrow E'$
    end if
  end for
  $\mathsf{M}[j] \leftarrow E_{\min}$
  $j \leftarrow j + 1$
end for
return M

Searching.

In the search phase, we wish to read enough q-grams such that the probability we must verify a window is small and the amount we can shift the window by is sufficiently large. We now recall some important lemmas from [29] that we will use in the analysis of our algorithm.

Lemma 98 ([29])**.** *The probability that two q-grams on Σ, $\sigma = |\Sigma|$, one being uniformly random, have a common subsequence of length $(1-c)q$ is at most $\frac{a\sigma^{-dq}}{q}$, where $a = (1+o(1))/(2\pi c(1-c))$ and $d = 1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)$. The probability decreases exponentially for positive d, which holds if $c < 1 - \frac{e}{\sqrt{\sigma}}$.*

Lemma 99 ([29])**.** *If s is a q-gram that occurs with less than cq differences in a given string u, $|u| \geq q$, then s has a common subsequence of length $q-cq$ with some q-gram of u.*

By Lemmas 98 and 99, we know that the probability of a random q-gram occurring in a string of length m with less than cq differences is no more than $ma\sigma^{-dq}/q$ as we have $m-q+1$ q-grams in the string. For circular string matching this is not sufficient. To ensure that we have the q-grams of all possible rotations of pattern x, we instead consider the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ and extract the q-grams from x'. We may have up to $2m-q$ q-grams, but to simplify the analysis we assume we have $2m$ and so the probability becomes $2ma\sigma^{-dq}/q$.

In the case when we read k/cq q-grams, we know that with probability at most $(k/cq)2ma\sigma^{-dq}/q$ we have found less than k differences. This does not permit us to throw out the window if all q-grams occur with at most cq differences. To fix this, we instead read $1+k/cq$ q-grams. If any q-gram occurs with less than cq differences, we will need to verify the window; but if they all occur with at least cq differences, we must exceed the threshold k and can shift the window. When shifting the window we have the case that we shift after verifying the window and the case that the differences exceed k so we do not verify the window. If we have verified the window, we can shift past the last position we checked for an occurrence: we can shift by $m-k$ positions. If we have not verified the window, as we read a fixed number of q-grams, we know the minimum-length shift we can make is one position past this point. The length of this shift is at least $m-k-(q+k/c)$ positions. This means we will have at most $\frac{n}{m-k-(q+k/c)} = \mathcal{O}(\frac{n}{m})$ windows. The previous statement is only true assuming $m-q > k+k/c$, as then the denominator is positive. From there we see that we also have the condition that $q+k/c$ can be at most ϵm, where $\epsilon < 1$, so the denominator will be $\mathcal{O}(m)$. This puts a slightly stricter condition on c, that is, $c > \frac{k}{\epsilon m-q-k}$.

From the above discussion, we can see that, for each window, we verify with probability no greater than $(1+k/cq)2ma\sigma^{-dq}/q$, where $a = (1+o(1))/(2\pi c(1-c))$ and $d = 1-c+2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)$. So the probability that a verification is triggered is

$$\frac{(1+k/cq)2ma\sigma^{-dq}}{q}.$$

By Lemma 96, the verification takes time $\mathcal{O}(m^2k)$, so, per window, we have an expected cost of

$$\frac{(1+k/cq)2ma\sigma^{-dq}\mathcal{O}(m^2k)}{q} = \mathcal{O}(\frac{(q+k)m^3ka\sigma^{-dq}}{q^2}).$$

We wish to ensure that the probability of verifying a window is small enough that the average work done is no more than the work we must do if we skip a window without verification. When we do not verify a window, we read $1 + k/cq$ q-grams and shift the window. This means that we read $q + k/c = \mathcal{O}(q + k)$ letters. So a sufficient condition is the following:

$$\frac{(q+k)m^3ka\sigma^{-dq}}{q^2} \leq \mathcal{O}(q+k).$$

Or equivalently the below expression, where f is the constant of proportionality:

$$\frac{(q+k)m^3ka\sigma^{-dq}}{q^2} \leq f(q+k).$$

By rearranging and setting $f = \sigma$ we get the condition on the value of q below. Note that setting $f = \sigma$ is simply for convenience as $\log_\sigma \sigma = 1$. Any value for $\log_\sigma f = \mathcal{O}(1)$ is sufficient and would simply lead to an additional constant in the analysis below, which may be removed in the same way we will deal with a.

$$q \geq \frac{3\log_\sigma m + \log_\sigma k + \log_\sigma a - 2\log_\sigma q}{d}.$$

From the condition on q we can see that it is sufficient to pick $q = \mathcal{O}(\log_\sigma km)$, so asymptotically on m we get the following:

$$q \geq \frac{3\log_\sigma m + \log_\sigma k - \mathcal{O}(\log_\sigma \log_\sigma km)}{d}.$$

Therefore, for sufficiently large m, the following condition is sufficient for optimality:

$$q = \frac{3\log_\sigma m + \log_\sigma k}{d}$$

$$q = \frac{3\log_\sigma m + \log_\sigma k}{1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)}.$$

For this analysis to hold we must be able to read the required number of q-grams to ensure the probability of verifying a window is small enough to negate the work of doing it. Not that the above probability is the probability that at least one of q-grams match with less than cq differences. To ensure we have enough unread random q-grams in the window for Lemma 99 to hold in the above analysis the window must be of size $m - k \geq 2q + 2k/c$. Now we consider the case where $2q + 2k/c > m - k \geq 2q + k/c$. If we have just verified a window then we have enough new random q-grams and our analysis holds. If we have just shifted then we know that all the q-grams we previously read matched with at least cq differences and we have between 1 and k/qc q-grams and the probability that one of these matches with less than cq difference is less than in the analysis above so it holds.

The condition $m - k \geq 2q + k/c$ implies a condition on c, it must be the case that $c \geq \frac{k}{m-k-2q}$. This condition on c is weaker than our previous condition on c, so to determine the *error ratio* $\frac{k}{m}$, we should use the stronger condition. Additionally, from Lemma 98, we know that $c < 1 - \frac{e}{\sqrt{\sigma}}$. So we must pick a value for c subject to $\frac{k}{\epsilon m-k-q} \leq c < 1 - \frac{e}{\sqrt{\sigma}}$. This inequality implies a limit on the error ratio for which our algorithm is optimal. Clearly it must be the case that $\frac{k}{\epsilon m-k-q} < 1 - \frac{e}{\sqrt{\sigma}}$ for $\epsilon < 1$. Rearranging the inequality implies the following sufficient condition on our error ratio:

$$\frac{2k}{m} < \epsilon - \frac{q}{m} - \frac{\epsilon e}{\sqrt{\sigma}} + \frac{qe}{m\sqrt{\sigma}} + \frac{ke}{m\sqrt{\sigma}}.$$

From here we can factorise and divide everything by two to get the following:

$$\frac{k}{m} < \frac{\epsilon}{2} - \frac{q}{2m} - \frac{e}{2\sqrt{\sigma}}(\epsilon - \frac{q}{m} - \frac{k}{m}).$$

So asymptotically on m we get the following:

$$\frac{k}{m} < \frac{\epsilon}{2} - \mathcal{O}(\frac{1}{\sqrt{\sigma}}).$$

Note that actually this technique can work for any ratio which satisfies the following:

$$\frac{k}{m} < \frac{1}{2} - \mathcal{O}(\frac{1}{\sqrt{\sigma}}).$$

As for any ratio below this, we can pick a large enough value for ϵ such that asymptotically on m our algorithm will work in the claimed search

time. By choosing a suitable value for c and $q \geq \frac{3\log_\sigma m + \log_\sigma k}{d}$ we obtain the following.

Theorem 100. *The problem ACSM can be solved in optimal average-case time* $\mathcal{O}(n(k + \log_\sigma m)/m)$.

7.7 Comparison with Existing Algorithms

To the best of our knowledge, the only other algorithms to achieve optimal average-case time for approximate circular string matching are the algorithms presented in [50] for multiple approximate string matching. These algorithms achieve optimality but have different preprocessing and space requirements. In this section, we analyse these results and compare them with our approach. We refer to the algorithm presented in Section 7.6 as BIP.

Applying the algorithms of [50] to approximate circular string matching requires us to reduce the problem to multiple approximate string matching for matching the $r = m$ rotations of x. For the first algorithm presented in [50], we get the following time complexity:

$$\mathcal{O}(n(k + \log_\sigma rm)/m).$$

Setting $r = m$ it is clear this becomes:

$$\mathcal{O}(n(k + \log_\sigma m)/m).$$

This result is valid under the conditions that $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$, $r = \mathcal{O}(\min(n^{1/3}/m^2, \sigma^{o(m)}))$, and we have $\mathcal{O}(\sigma^q)$ space available, where q is subject to the constraint:

$$q \geq \frac{4\log_\sigma m + 2\log_\sigma r}{d}.$$

Again by setting $r = m$ this becomes:

$$q \geq \frac{6\log_\sigma m}{d}.$$

The preprocessing time of this algorithm is $\mathcal{O}(\sigma^q m^2)$. We will refer to this algorithm as FN1. The second algorithm, presented in [50], has the same preprocessing cost and requires space $\mathcal{O}(\sigma^q m)$. We will refer to this algorithm

Table 7.4: Comparison of optimal average-case algorithms for approximate circular string matching

Algorithm	Error Ratio (k/m)	Space	Preprocessing Time	Condition on q
FN1	$\frac{1}{2} - \mathcal{O}(\frac{1}{\sqrt{\sigma}})$	$\mathcal{O}(\sigma^q)$	$\mathcal{O}(\sigma^q m^2)$	$\frac{6\log_\sigma m}{d}$
FN2	$\frac{1}{2} - \mathcal{O}(\frac{1}{\sqrt{\sigma}})$	$\mathcal{O}(\sigma^q m)$	$\mathcal{O}(\sigma^q m^2)$	$\frac{4\log_\sigma m + \log_\sigma(m+\log_2 m)}{d}$
BIP	$\frac{1}{2} - \mathcal{O}(\frac{1}{\sqrt{\sigma}})$	$\mathcal{O}(\sigma^q)$	$\mathcal{O}(\sigma^q m q)$	$\frac{3\log_\sigma m + \log_\sigma k}{d}$

as FN2. The important difference between the two comes in the condition on q which is slightly lower for FN2:

$$q \geq \frac{3\log_\sigma m + \log_\sigma r + \log_\sigma(m + \log_2 r)}{d}.$$

Again by setting $r = m$ this becomes:

$$q \geq \frac{4\log_\sigma m + \log_\sigma(m + \log_2 m)}{d}.$$

To simplify the comparison between these approaches, we will ignore the factor of $\log_2 m$, and simply say that the value of q for algorithm FN2 is greater than or equal to $\frac{5\log_\sigma m}{d}$. This is lower than the sufficient requirement, so any saving we make using this value must be at least as good or better in reality.

First let us consider FN1. The preprocessing requirement of BIP is $\mathcal{O}(\sigma^q m q)$, so before any savings made due to the value of q for BIP, we have reduced the preprocessing cost by a factor of $\mathcal{O}(\frac{m}{q})$. Given the condition on q for BIP, it is clear to see that even in the worst-case, when $k = \mathcal{O}(m)$, BIP will make a saving of at least $2\log_\sigma m$ on the value of q. This corresponds to an additional saving of $\mathcal{O}(m^2)$ in preprocessing time bringing the total to $\mathcal{O}(\frac{m^3}{q})$ and $\mathcal{O}(m^2)$ in space. In the case of FN2, we make a saving of at least $\log_\sigma m$ on the value of q. This corresponds to a total saving of $\mathcal{O}(\frac{m^2}{q})$ in preprocessing time and $\mathcal{O}(m^2)$ in space. It should be noted that this is a fairly pessimistic analysis of the savings as we assumed $k = \mathcal{O}(m)$ and the exact savings we make depend on the value of d that is chosen. In this analysis, we assumed that $d = 1$, although this is not possible and it must be smaller. Due to this, any savings stated above can be considered conservative estimates. Table 7.4 corresponds to the above analysis.

7.8 Conclusions and Future Work

In this chapter, we presented new average-case algorithms for exact and approximate circular string matching. Algorithm ECSMF for exact circular string matching requires average-case time $\mathcal{O}(n)$; and Algorithms ACSMF and ACSMF-Simple for approximate circular string matching with k-mismatches require time $\mathcal{O}(n)$ for moderate values of k, that is $k = \mathcal{O}(m/\log_\sigma m)$. We showed how the same results can be easily obtained under the edit distance model. The presented algorithms were also implemented as library functions. Experimental results demonstrate that the functions provided in this library accelerate the computations by more than three orders of magnitude compared to a naïve approach.

For future work, we will explore the possibility of optimising our algorithms and the corresponding library implementation for the approximate case by using lossless filters for eliminating a possibly large fraction of the input that is guaranteed not to contain any approximate occurrence, such as [105] for the Hamming distance model or [106] for the edit distance model.

We have also presented a new average-case optimal algorithm for approximate circular string matching under the edit distance model. To the best of our knowledge, this algorithm is the first average-case optimal algorithm specifically designed for this problem. Other average-case optimal algorithms exist for this problem, but with higher preprocessing and space requirements than the presented algorithm. Additionally the considered problem is solved in a more direct fashion, that is, we avoid the reduction to multiple approximate string matching, and take greater advantage of the similarity of the rotations of the pattern.

A drawback of the presented technique is the complicated verification and preprocessing scheme which may not lead to a very efficient runtime in practical terms. However, this can be completely removed and the standard dynamic programming algorithm can be used instead with runtime $\mathcal{O}(m^3)$ for verification and $\mathcal{O}(\sigma^q m q^2)$ for preprocessing. The speed-ups mentioned in the previous section remain significant for this as we assumed that $k = \mathcal{O}(m)$. So even without the complicated verification and preprocessing scheme, we achieve a preprocessing speed up of at least $\mathcal{O}(m^2)$ and $\mathcal{O}(m)$ against FN1 and FN2, respectively. For future work, we plan on tackling the problem of multiple approximate circular string matching.

8

Concluding Remarks

In this chapter we summarise the contributions and open problems presented in the previous chapters of this thesis and suggest directions for future work.

In Chapter 3 we presented algorithm `GapsMis` and `GapsMis-L` for pairwise sequence alignment with a bounded number of gaps. The algorithms both take time $\mathcal{O}(nk\ell)$ time, one open problem is if it is possible to reduce this runtime. An approach to compute the alignment more efficiently may be to find an algorithm for computing the alignment with *exactly* ℓ gaps and then performing a binary search on the number of gaps. Another avenue for future work is to improve the implementation of `GapsMis` and `GapsMis-L` and port them to GPU based architectures.

In Chapter 4 we have presented $\mathcal{O}(kn)$ algorithms for the computation of the prefix table under both Hamming and edit distance. This allows for a compact representation of the approximate borders of string, something not possible when using the border array. We focus on the application of merging paired-end reads resulting from paired-end sequences, however, we also show how these simple data structures can be used for various problems important in a diverse range of applications. An avenue for future work would be to develop a tool for read merging based on the presented data structure.

In Chapter 5 we presented algorithms for the computation of inverted repeats in weighted strings. We present algorithms for the computation of exact and approximate inverted repeats under hamming distance taking $\mathcal{O}(n)$ and $\mathcal{O}(kn)$ respectively. Some of the techniques presented may also be more generally applicable for other problems in weighted strings. In particular we present an approach to efficiently compute the probability of any factor of a weighted string more general and with an exponentially smaller constant than existing methods. For future work we intend to work on the problem of inverted repeats under edit distance.

In Chapter 5 we also presented an $\mathcal{O}(n \log n)$ algorithm for the compu-

tation of all repetitions in a weighted sequence. This algorithm improves on the best known existing algorithm for repetitions in weighted strings which takes $\mathcal{O}(n \log n)$. Although our algorithm is optimal, it is known that repetitions as we have defined them are not the most compact representation of the repetitive elements in a strings. The computation of maximal periodicities or *runs* allows this information to be implicitly represented and reported in only $\mathcal{O}(n)$ time in regular string. For future work we plan to investigate the efficient computation of runs in weighted strings. Finally we have shown an alternative optimal algorithm for the computation of repetitions and show how this can be used to compute covers in strings more efficiently than existing algorithms. For future work we will attempt to use our technique to improve the computation of seeds.

In Chapter 6 fast average-case algorithms for wildcard matching were presented. A number of different models were considered and these results imply the existence of difficult patterns, but that such patterns are actually quite rare. We have derived a lower bound on the complexity for the problems considered and also showed that a fixed inspection scheme will never lead to an optimal algorithm, but a greedy inspection scheme will. The area of average-case algorithms for wildcards is very undeveloped so many interesting open questions still exist.

- What is a tight bound on the average-case complexity of wildcard matching?
- What is the average-case complexity of wildcard matching if preprocessing is restricted to $\mathcal{O}(m^c)$ for some constant $c > 0$?
- What is the average-case complexity of wildcard matching if the space is restricted to $\mathcal{O}(m^c)$ for some constant $c > 0$?

In Chapter 7 we have presented a number of algorithms for circular string matching. For the problem of circular pattern matching under Hamming distance we design a linear on average algorithm which is also shown to be practically efficient as well as fast on average. We then consider the problem of circular string matching under edit distance and present and average-case optimal algorithm for this problem which reduces the time and space complexity when compared with existing algorithms. We then note that the Hamming distance algorithm can be easily applied to edit distance and vice versa. For future work we plan on studying the problem of multiple approximate string circular string matching.

Bibliography

[1] Nomenclature committee of the international union of biochemistry (NC-IUB). nomenclature for incompletely specified bases in nucleic acid sequences. recommendations 1984. *Eur J Biochem*, 150(1):1–5, 1985.

[2] Karl Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[3] Tatsuya Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[4] Nikolaos Alachiotis, Simon Berger, Tomas Flouri, Solon P. Pissis, and Alexandros Stamatakis. libgapmis: an ultrafast library for short-read single-gap alignment. In *Proceedings of the International Conference on Bioinformatics & Biomedicine Workshops (BIBMW 2012)*, pages 688–695, 2012.

[5] Nikolaos Alachiotis, Simon Berger, Tomas Flouri, Solon P. Pissis, and Alexandros Stamatakis. libgapmis: extending short-read alignments. *BMC Bioinformatics*, 14:S4, 2013.

[6] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[7] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. In ACM-SIAM, editor, *Proceedings of the eleventh annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 794–803, USA, 2000. Society for Industrial and Applied Mathematics.

[8] Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.

[9] Ricardo Baeza-Yates. Algorithms for string searching. *SIGIR Forum*, 23(3-4):34–58, April 1989.

[10] Ricardo Baeza-Yates and Gonzalo Navarro. New and faster filters for multiple approximate string matching. In *Random Structures and Algorithms (RSA)*, page 2002, 1998.

[11] Shankar Balasubramanian, David Klenerman, Colin Barnes, and Michael A. Osborne. Patent US20077232656, 2007.

[12] Carl Barton, Tomáš Flouri, Costas S. Iliopoulos, and Solon P. Pissis. Gapsmis: Flexible sequence alignment with a bounded number of gaps. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 402:402–402:411, New York, NY, USA, 2013. ACM.

[13] Carl Barton, Tom Flouri, Costas S. Iliopoulos, and Solon P. Pissis. Global and local sequence alignment with a bounded number of gaps. *Theoretical Computer Science*, 582(0):1 – 16, 2015.

[14] Carl Barton and Costas S. Iliopoulos. On the average-case complexity of pattern matching with wildcards. *CoRR*, abs/1407.0950, 2014.

[15] Carl Barton, Costas S. Iliopoulos, Inbok Lee, Laurent Mouchard, Kunsoo Park, and Solon P. Pissis. Extending alignments with k-mismatches and ℓ-gaps. *Theoretical Computer Science*, 525:80–88, 2014.

[16] Carl Barton, Costas S. Iliopoulos, Nicola Mulder, and Bruce Watson. Identification of all exact and approximate inverted repeats in regular and weighted sequences. In *Engineering Applications of Neural Networks - 14th International Conference, EANN 2013, Halkidiki, Greece, September 13-16, 2013 Proceedings, Part II*, pages 11–19, 2013.

[17] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Circular string matching revisited. In *Proceedings of the Fourteenth Italian Conference on Theoretical Computer Science (ICTCS 2013)*, pages 200–205, 2013.

[18] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9(9), 2014.

[19] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Optimal computation of all tandem repeats in a weighted sequence. *Algorithms for Molecular Biology*, 9(21), 2014.

[20] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Average-case optimal approximate circular string matching. In A.-H. Dediu, E. Formenti, C. Martn-Vide, and B Truthe, editors, *Language and Automata Theory and Applications*, volume 8977 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin Heidelberg, 2015.

[21] Carl Barton, Costas S. Iliopoulos, Solon P. Pissis, and W. F. Smyth. Fast & simple computations using prefix tables under hamming and edit distance. In *Combinatorial Algorithms*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014. (in press).

[22] Carl Barton and Solon P. Pissis. Optimal computation of all repetitions in a weighted string. In Costas S. Iliopoulos and Alessio Langiu, editors, *International Conference on Algorithms for Big Data (ICABD2014)*, number 1146 in CEUR-WS Proceedings, pages 9–15, Aachen, 2014.

[23] Gary Benson. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research*, 27(2):573–580, 1999.

[24] Philip Bille, Inge Li Grtz, Hjalte Wedel Vildhj, and Sren Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 55(1):41–60, 2014.

[25] Valentina Boeva, Mireille Regnier, Dmitri Papatsenko, and Vsevolod Makeev. Short fuzzy tandem repeats in genomic sequences, identification, and possible role in regulation of gene expression. *Bioinformatics*, 22(6):676–684, 2006.

[26] Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.

[27] Michael Brown and Charles Wilson. RNA pseudoknot modeling using intersections of stochastic context free grammars with applications to database search. In *Pacific Symposium on Biocomputing*, pages 109–125, 1995.

[28] Terence A. Brown. *Genomes*. BIOS Scientific Publishers, London, UK, 1999.

[29] William I. Chang and Thomas G. Marr. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, CPM '94, pages 259–273, London, UK, 1994. Springer-Verlag.

[30] Jiunn-Liang Chen and Carol W. Greider. Functional analysis of the pseudoknot structure in human telomerase RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 102(23):8080–8085, 2005.

[31] Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *Computer Journal*, 2013.

[32] Manolis Christodoulakis, Costas S. Iliopoulos, Laurent Mouchard, Katerina Perdikuri, Athanasios K. Tsakalidis, and Kostas Tsichlas. Computation of repetitions and regularities of biologically weighted sequences. *Journal of Computational Biology*, 13(6):1214–1231, 2006.

[33] Manolis Christodoulakis, Costas S. Iliopoulos, Laurent Mouchard, and Kostas Tsichlas. Pattern matching on weighted sequences. In *Proceedings of the Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets)*, pages 17–30, 2004.

[34] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, January 2007.

[35] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pages 91–100, New York, NY, USA, 2004. ACM.

[36] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 592–601, New York, NY, USA, 2002. ACM.

[37] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[38] Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4-5):247–267, 1994.

[39] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[40] Maxime Crochemore, Lucian Ilie, and Wojciech Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227 – 5235, 2009.

[41] Shiri Dori and Gad M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Information Processing Letters*, 98(2):66–72, 2006.

[42] Francisco Fernandes, Lusa Pereira, and Ana T Freitas. CSA: An efficient algorithm to improve circular DNA multiple alignment. *BMC Bioinformatics*, 10(1):1–13, 2009.

[43] Johannes Fischer. Inducing the LCP-Array. In Frank Dehne, John Iacono, and Jrg-Rdiger Sack, editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer Berlin Heidelberg, 2011.

[44] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[45] Michael J. Fischer and Michael S. Paterson. String-matching and other products. Technical report, Cambridge, MA, USA, 1974.

[46] Tomás Flouri, Costas S. Iliopoulos, Kunsoo Park, and Solon P. Pissis. GapMis-OMP: Pairwise short-read alignment on multi-core architectures. In Lazaros S. Iliadis, Ilias Maglogiannis, Harris Papadopoulos, Kostas Karatzas, and Spyros Sioutas, editors, *AIAI (2)*, volume 382 of *IFIP Advances in Information and Communication Technology*, pages 593–601. Springer, 2012.

[47] Tomáš Flouri, Kimon Frousios, Costas S. Iliopoulos, Kunsoo Park, Solon P. Pissis, and German Tischler. GapMis: a tool for pairwise sequence alignment with a single gap. *Recent Patents on DNA and Gene Sequence*, 7:84–95, 2013.

[48] Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009.

[49] Kimmo Fredriksson and Gonzalo Navarro. Average-optimal multiple approximate string matching. In Ricardo Baeza-Yates, Edgar Chvez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 109–128. Springer Berlin Heidelberg, 2003.

[50] Kimmo Fredriksson and Gonzalo Navarro. Average-optimal single and multiple approximate string matching. *Journal of Experimental Algorithmics*, 9, December 2004.

[51] Kimmo Fredriksson and Gonzalo Navarro. Flexible music retrieval in sublinear time. In *Proceedings of the 10th Prague Stringology Conference (PSC'05)*, pages 174–188, 2005.

[52] Kimon Frousios, Costas S. Iliopoulos, Laurent Mouchard, Solon P. Pissis, and German Tischler. REAL: an efficient REad ALigner for next generation sequencing reads. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 154–159, USA, 2010. ACM.

[53] Zvi Galil and Kunsoo Park. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.

[54] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. *New indices for text: PAT Trees and PAT arrays*, pages 66–82. Prentice-Hall, Inc., 1992.

[55] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705 – 708, 1982.

[56] Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010.

[57] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[58] Henry S. Hall and Samuel R. Knight. *Higher Algebra*. MacMillan, 1950.

[59] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[60] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressed text indexing with wildcards. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval*, volume 7024 of *Lecture Notes in Computer Science*, pages 267–277. Springer Berlin Heidelberg, 2011.

[61] Ping-Hui Hsu, Kuan-Yu Chen, and Kun-Mao Chao. Finding all approximate gapped palindromes. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, ISAAC '09, pages 1084–1093, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Edward David Hyman. A new method of sequencing DNA. *Analytical Biochemistry*, 174(2):423 – 436, 1988.

[63] Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, 2010.

[64] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae.*, 71(2,3):259–277, February 2006.

[65] Costas S. Iliopoulos, Laurent Mouchard, Katerina Perdikuri, and Athanasios K. Tsakalidis. Computing the repetitions in a biological weighted sequence. *Journal of Automata, Languages and Combinatorics*, 10(5/6):687–696, 2005.

[66] Costas S. Iliopoulos, Katerina Perdikuri, Evangelos Theodoridis, Athanasios Tsakalidis, and Kostas Tsichlas. Motif extraction from weighted sequences. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE), volume 3246 of LNCS*, pages 286–297. Springer, 2004.

[67] Costas S. Iliopoulos and M. Sohel Rahman. Pattern matching algorithms with dont cares. In *Proceedings of 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 2, pages 116–126, 2007.

[68] Costas S. Iliopoulos and M. Sohel Rahman. Indexing circular patterns. In *Proceedings of the 2nd International Conference on Algorithms and Computation*, WALCOM'08, pages 46–57, Berlin, Heidelberg, 2008. Springer-Verlag.

[69] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Proceedings of the 39th Symposium on Foundations of Computer Science*, pages 166–173, 1998.

[70] Jingyue Ju, Zengmin Li, John R. Edwards, and Yashuhiro Itagaki. Patent EP1790736, 2007.

[71] Adam Kalai. Efficient pattern-matching with dont cares. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 655–656, 2002.

[72] Cyriac Kandoth, Fikret Ercal, and RonaldL Frank. A framework for automated enrichment of functionally significant inverted repeats in whole genomes. *BMC Bioinformatics*, 11:1–10, 2010.

[73] Yuki Kato, Hiroyuki Seki, and Tadao Kasami. Stochastic multiple context-free grammar for RNA pseudoknot modeling. In *Proceedings of the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms*, TAGRF '06, pages 57–64, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[74] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[75] Roman Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*, 31(13):3672–3678, 2003.

[76] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Symposium on Foundations of Computer Science*, pages 596–604. IEEE Computer Society, 1999.

[77] Roman Kolpakov and Gregory Kucherov. Finding approximate repetitions under hamming distance. *Theoretical Computer Science*, 303(1):135 – 156, 2003. Logic and Complexity in Computer Science.

[78] Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with Dont cares. In Takeshi Tokuyama, editor, *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 846–857. Springer Berlin Heidelberg, 2007.

[79] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal of Computing*, 27(2):557–582, 1998.

[80] Gad M. Landau and Uzi Vishkin. Efficient string matching in the presence of errors. In IEEE, editor, *Proceedings of the twenty-sixth annual Symposium on Foundations of Computer Science (FOCS 1985)*, pages 126–136, USA, 1985. IEEE Computer Society.

[81] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 4(9):357–359, 2013.

[82] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25+, 2009.

[83] Taehyung Lee, Joong Chae Na, Heejin Park, Kunsoo Park, and Jeong Seop Sim. Finding optimal alignment and consensus of circular strings. In *Proceedings of the 21st annual Conference on Combinatorial Pattern Matching*, CPM'10, pages 310–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[84] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, Soviet Physics Doklady, 1966.

[85] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[86] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(16):1966–1967, 2009.

[87] Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.

[88] Jie Lin and Don Adjeroh. All-Against-All Circular Pattern Matching. *Computer Journal*, 55(7):897–906, 2012.

[89] M. Lothaire, editor. *Applied Combinatorics on Words*. Cambridge University Press, 2005.

[90] Michael G. Main and Richard J. Lorentz. An $\mathcal{O}(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.

[91] Glenn Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351, July 1975.

[92] Elliot H. Margulies and Ewan Birney. Approaches to comparative sequence analysis: towards a functional view of vertebrate genomes. *Nature Reviews Genetics*, 9(4):303–313, 2008.

[93] Andre E. Minosche, Juliane C. Dohm, and Heinz Himmelbauer. Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and Genome Analyzer systems. *Genome Biology*, 12:R112, 2011.

[94] Michael Mitas. Trinucleotide repeats associated with human disease. *Nucleic Acids Research*, 25(12):2245–2253, 1997.

[95] Dennis Moore and William F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, pages 511–515, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[96] Axel Mosig, Ivo L. Hofacker, and Peter F. Stadler. Comparative Analysis of Cyclic Sequences: Viroids and other Small Circular RNAs. In Daniel H. Huson, Oliver Kohlbacher, Andrei N. Lupas, Kay Nieselt, and Andreas Zell, editors, *German Conference on Bioinformatics*, volume 83 of *LNI*, pages 93–102. GI, 2006.

[97] National Center for Biotechnology Information (NCBI). `ftp://ftp.ncbi.nih.gov/blast/matrices/NUC.4.4`, April 2013.

[98] National Center for Biotechnology Information (NCBI). `ftp://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62`, April 2013.

[99] National Center for Biotechnology Information (NCBI). `http://www.ncbi.nlm.nih.gov/`, April 2013.

[100] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, USA, 2002.

[101] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[102] Sarah B. Ng, Emily H. Turner, Peggy D. Robertson, Steven D. Flygare, Abigail W. Bigham, Choli Lee, Tristan Shaffer, Michelle Wong, Arindam Bhattacharjee, Evan E. Eichler, Michael Bamshad, Deborah A. Nickerson, and Jay Shendure. Targeted capture and massively parallel sequencing of 12 human exomes. *Nature*, 461(7261):272–276, 2009.

[103] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, Washington, DC, USA, 2009. IEEE Computer Society.

[104] Pia Ostergaard, Michael A Simpson, Glen Brice, Sahar Mansour, Fiona C Connell, Alexandros Onoufriadis, Anne H Child, Jae Hwang, Kamini Kalidas, Peter S Mortimer, Richard Trembath, and Steve Jeffery. Rapid identification of mutations in GJC2 in primary lymphoedema using whole exome sequencing combined with linkage analysis with delineation of the phenotype. *Journal of Medical Genetics*, 48(4):251–255, 2010.

[105] Pierre Peterlongo, Nadia Pisanti, Frdric Boyer, Alair Pereira do Lago, and Marie-France Sagot. Lossless filter for multiple repetitions with hamming distance. *Journal of Discrete Algorithms*, 6(3):497 – 509, 2008.

[106] Pierre Peterlongo, Gustavo A. Sacomoto, Pereira, Nadia Pisanti, and Marie-France Sagot. Lossless filter for multiple repeats with bounded edit distance. *Algorithms for Molecular Biology*, 4(1):3+, January 2009.

[107] Ron Y. Pinter. Efficient string matching with dont-care patterns. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series*, pages 11–29. Springer Berlin Heidelberg, 1985.

[108] Pizza&Chili. `http://pizzachili.dcc.uchile.cl/`, April 2013.

[109] Comelis W. A. Pleij, Krijn Rietveld, and Leendert Bosch. A new principle of RNA folding based on pseudoknotting. *Nucleic Acids Research*, 13(5):1717–1731, 1985.

[110] Alexandre H.L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35(11):2581 – 2591, 2002.

[111] Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, 2000.

[112] Ronald L. Rivest. Partial-Match Retrieval Algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[113] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.

[114] Jonathan M. Rothberg, Joel S. Bader, Scott B. Dewell, Keith E. McDade, John W. Simpson, Jan Berka, and Christopher M. Colangelo. Founding patent of 454 life sciences. Patent US20077211390, 2007.

[115] F. Sanger, G. M. Air, B. G. Barrell, Nigel L. Brown, A. R. Coulson, J. C. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage ϕ X174 DNA. *Nature*, 265(5596):687–695, 1977.

[116] Frederick Sanger and Alan R. Coulson. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *Journal of Molecular Biology*, 94(3):441 – 448, 1975.

[117] Frederick Sanger, Steve Nicklen, and Alan R Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of The National Academy of Sciences of The United States Of America*, 74(12):5463–5467, 1977.

[118] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[119] Jay Shendure, Gregory J. Porreca, Nikos B. Reppas, Xiaoxia Lin, John P. McCutcheon, Abraham M. Rosenbaum, Michael D. Wang, Kun Zhang, Robi D. Mitra, and George M. Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science*, 309(5741):1728–1732, September 2005.

[120] Michael A. Simpson, Melita D. Irving, Esra Asilmaz, Mary J. Gray, Dimitra Dafou, Frances V. Elmslie, Sahar Mansour, Sue E. Holder, Caroline E. Brain, Barbara K. Burton, Katherine H. Kim, Richard M. Pauli, Salim Aftimos, Helen Stewart, Chong Ae Kim, Muriel Holder-Espinasse, Stephen P. Robertson, William M. Drake, and Richard C. Trembath. Mutations in NOTCH2 cause Hajdu-Cheney syndrome, a disorder of severe and progressive bone loss. *Nature Genetics*, 43(4):303–305, 2011.

[121] Martin Simunek and Borivoj Melichar. Borders and finite automata. *Int. J. Found. Comput. Sci.*, 18(4):859–871, 2007.

[122] William F. Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, 2003.

[123] William F. Smyth and Shu Wang. New perspectives on the prefix array. In *Proceedings of the 15th String Processing & Information Retrieval Symposium*, volume 5280 of *Lecture Notes in Computer Science*, pages 133–143. Springer, 2008.

[124] StringPedia. `http://stringpedia.bsmithers.co.uk`, April 2013.

[125] Chris Thachuk. Succincter text indexing with wildcards. In Raffaele Giancarlo and Giovanni Manzini, editors, *Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 27–40. Springer Berlin Heidelberg, 2011.

[126] Esko Ukkonen. On approximate string matching. In Marek Karpinski, editor, *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 487–495. Springer Berlin Heidelberg, 1983.

[127] Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. In *Proceedings of the 21st annual conference on Combinatorial pattern matching*, CPM'10, pages 76–87. Springer-Verlag, 2010.

[128] Alex van Belkum, Stewart Scherer, Loek van Alphen, and Henri Verbrugh. Short-sequence DNA repeats in prokaryotic genomes. *Microbiol. Mol. Biol. Rev.*, 62(2):275–293, 1998.

[129] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[130] Michael S. Waterman and Temple F. Smith. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[131] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[132] Andrew Chi-Chih Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.

[133] Hui Zhang, Qing Guo, Jing Fan, and Costas S. Iliopoulos. Loose and strict repeats in weighted sequences of proteins. *Protein and Peptide Letters*, 17(9):1136–1142, 2010.

[134] Hui Zhang, Qing Guo, and Costas S Iliopoulos. Varieties of regularities in weighted sequences. In *Algorithmic Aspects in Information and Management*, pages 271–280. Springer, 2010.

[135] Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Locating tandem repeats in weighted sequences in proteins. *BMC Bioinformatics*, 14(S-8):S2, 2013.

[136] Jiajie Zhang, Kassian Kobert, Tomáš Flouri, and Alexandros Stamatakis. PEAR: A fast and accurate Illumina Paired-End reAd mergeR. *Bioinformatics*, 2013.